

Unclassified

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

4

AD-A196 931

PII Redacted

REPORT DOCUMENTATION PAGE		READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER	2. GOVT ACCESSION NO.	3. RECIPIENT'S CATALOG NUMBER
4. TITLE (and Subtitle) Parallel Programming Paradigms		5. TYPE OF REPORT & PERIOD COVERED Dissertation-Technical Report
		6. PERFORMING ORG. REPORT NUMBER
7. AUTHOR(s) Philip Arne Nelson		8. CONTRACT OR GRANT NUMBER(s) N00014-86-K-0264
9. PERFORMING ORGANIZATION NAME AND ADDRESS University of Washington Department of Computer Science Seattle, Washington 98195		10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS
11. CONTROLLING OFFICE NAME AND ADDRESS Office of Naval Research Information Systems Program Arlington, VA 22217		12. REPORT DATE July 1987
		13. NUMBER OF PAGES 132
14. MONITORING AGENCY NAME & ADDRESS (if different from Controlling Office)		15. SECURITY CLASS. (of this report) Unclassified
		15a. DECLASSIFICATION/DOWNGRADING SCHEDULE
16. DISTRIBUTION STATEMENT (of this Report) Distribution of this report is unlimited.		
17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)		
18. SUPPLEMENTARY NOTES		
19. KEY WORDS (Continue on reverse side if necessary and identify by block number) MIMD machines; parallel algorithms; compute-aggregate-broadcast; divide and conquer; pipe-lining; reduction; contraction problem; matrix multiplication; topological sort		
20. ABSTRACT (Continue on reverse side if necessary and identify by block number) Paradigms for the development of sequential algorithms, such as divide-and-conquer and the greedy method, are well known. Paradigms for the development of parallel algorithms, especially algorithms for non-shared memory MIMD machines, are not well known. These paradigms are important, not only as tools for the development of new algorithms, but also because algorithms using the same paradigm often have common properties that can be exploited by operations such as contraction. This dissertation identifies four primary paradigms used by non-shared memory		

DTIC
ELECTE
JUL 07 1988
S & D

DD FORM 1 JAN 73 1473

EDITION OF 1 NOV 65 IS OBSOLETE

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

MIMD algorithms. They are compute-aggregate-broadcast, divide-and-conquer pipelining, and reduction. Compute-aggregate-broadcast is used, for example, in numerical approximation algorithms like the conjugate gradient iterations. Three variations of the compute-aggregate-broadcast paradigm are studied. Divide-and-conquer is shown to be applicable to parallel algorithms. The relationship between divide-and-conquer algorithms and the n-cube is studied. Systolic techniques are known to be broadly applicable for the development of MIMD algorithms. Systolic algorithms are shown to be members of the more general pipelining paradigm. Finally, the reduction paradigm is briefly studied.

Accession For	
NTIS ORIGIN	<input checked="" type="checkbox"/>
DTIC TAG	<input type="checkbox"/>
Unannounced	<input type="checkbox"/>
Justification	
By	
Distribution	
Availability Codes	
Dist	Availability Codes
A-1	



Parallel Programming Paradigms

Philip Arne Nelson
Department of Computer Science, FR-35
University of Washington
Seattle, Washington 98195

TR 87-07-02
July 1987

A dissertation submitted in partial fulfillment of the requirements for the degree of Doctor of Philosophy, University of Washington.

This research was supported in part by National Science Foundation grant DCR-8416878 and by the Office of Naval Research Contracts No. N00014-86-K-0264 and No. N00014-85-K-0328.

Parallel Programming Paradigms

Philip Arne Nelson

University of Washington

Abstract

Paradigms for the development of sequential algorithms, such as divide-and-conquer and the greedy method, are well known. Paradigms for the development of parallel algorithms, especially algorithms for non-shared memory MIMD machines, are not well known. These paradigms are important, not only as tools for the development of new algorithms, but also because algorithms using the same paradigm often have common properties that can be exploited by operations such as contraction.

This dissertation identifies four primary paradigms used by non-shared memory MIMD algorithms. They are compute-aggregate-broadcast, divide-and-conquer, pipelining, and reduction. Compute-aggregate-broadcast is used, for example, in numerical approximation algorithms like the conjugate gradient iterations. Three variations of the compute-aggregate-broadcast paradigm are studied. Divide-and-conquer is shown to be applicable to parallel algorithms. The relationship between divide-and-conquer algorithms and the n-cube is studied. Systolic techniques are known to be broadly applicable for the development of MIMD algorithms. Systolic algorithms are shown to be members of the more general pipelining paradigm. Finally, the reduction paradigm is briefly studied.

The contraction problem, the problem arising when an algorithm requires more processors than are available on the execution machine, is studied. Special attention is given to common solutions to the contraction problem in each paradigm. An optimal contraction is given for algorithms using the tree interconnection structure. For other contractions, a method for comparing them is given. (KR) ↑

This dissertation also presents two new parallel algorithms. The first algorithm is a divide-and-conquer matrix multiplication algorithm with time complexity of $O(n)$ using $O(n^2)$ processors and $O(\log n)$ using $O(n^3)$ processors. The second algorithm is a compute-aggregate-broadcast topological sort with time complexity of $O(n \log n)$ using $O(n^2)$ processors and $O(\log^2 n)$ using $O(n^3)$ processors.

Table of Contents

	<i>Page</i>
CHAPTER 1: Parallel Programming	1
1.1. Parallel Machines and the Type Architecture	2
1.2. Programming Paradigms	4
1.3. Contraction	6
1.4. Parallel Algorithms	13
CHAPTER 2: The Compute-Aggregate-Broadcast Paradigm	17
2.1. CAB paradigm and Jacobi Iterations	17
2.2. Numerical CAB	20
2.2.1. SOR	20
2.2.2. Integration	22
2.3. Non-numerical CAB	23
2.3.1. Topological Sort	24
2.3.2. Hitech Chess	30
2.3.3. Production Based Expert Systems	32
2.4. A Method for Improving Speed	34
2.5. Contraction of CAB Algorithms	35
2.5.1. Tree Contraction	35
2.5.2. Multi-phase Contraction	40
CHAPTER 3: The Divide-and-Conquer Paradigm	42
3.1. Batcher's Bitonic Sort	43
3.2. Matrix Multiplication	44
3.3. The Fast Fourier Transform	49
3.4. Divide-and-conquer and the n-cube	52
3.5. Connected Ones	55
3.6. Contraction of Divide-and-Conquer Algorithms	56
CHAPTER 4: The Pipelining and Systolic Paradigms	61
4.1. Pipeline and Systolic Definitions	61
4.2. Flow Test	62
4.3. Systolic Algorithms	63
4.3.1. Band Matrix Multiply	63
4.3.2. WAP Matrix Multiply	65
4.3.3. Dynamic Programming	66

4.3.4. Lower Triangular Linear System Solver	67
4.4. Other Algorithms	68
4.4.1. Vector Sum Algorithm	68
4.4.2. Hough Transform	69
4.4.3. Funneled Pipelines	72
4.5. Contraction of Pipeline Algorithms	73
CHAPTER 5: Other Paradigms	77
5.1. Reduction	77
5.2. Arbitrary Communication Algorithms	79
CHAPTER 6: Summary and Further Research	81
REFERENCES:	85
APPENDIX A: Implementations of Selected Algorithms	94
A.1. The Poker Programming Environment	94
A.2. Jacobi Iterations	100
A.3. Batcher's Sort	106
A.4. Matrix Multiply	109
A.5. WAP Matrix Multiply	111
A.6. Contractions	116
A.6.1. Folded Tree Algorithm	116
A.6.2. Leiserson Layout Tree Algorithm	120
A.6.3. Coalesced Mesh Algorithm	124
A.6.4. Folded Mesh Algorithm	130

List of Figures

<i>Number</i>	<i>Page</i>
1-1. A 15 Node Tree	8
1-2. A 5 Processor Contraction	9
1-3. A 4 Processor Contraction	9
1-4. An 8 Processor Contraction of a 31 Node Tree	10
1-5. An Algorithm with Sequential Dependencies	11
1-6. Maximum Algorithm	15
2-1. Electric Field Problem	18
2-2. 16 Process Jacobi	19
2-3. Red/Black SOR	21
2-4. UpdateLevel Algorithm, Non-root Process	25
2-5. UpdateLevel Algorithm, Root Process	26
2-6. The Mesh of Trees with Roots on the Diagonal	27
2-7. Topological Sort Algorithm	29
2-8. CAB in Hitech Chess	31
2-9. DADO Processor Structure	33
2-10. Tree Contractions	36
2-11. Leiserson Tree Construction	36
2-12. Berman and Snyder Tree Contraction	37
3-1. Bitonic Sort Data Exchanges	43
3-2. 2×2 Product and Communication Structure	45
3-3. 4×4 Connections	46
3-4. 8×8 Connections for $O(\log n)$ Matrix Product	47
3-5. Sequential FFT Algorithm	50
3-6. Parallel FFT data movement	51
3-7. Single Process, Parallel FFT Algorithm	53
3-8. DAC Integrate Connections	54
3-9. Connected Ones Communication	56
3-10. An Order 4 N-cube	57
3-11. Batcher's Bitonic Merge Sort	59
4-1. Band Matrix Multiply	64
4-2. Data Staging for the WAP Matrix Multiply	65
4-3. Dynamic Programming Process Structure	66
4-4. Systolic Linear System Solver	68
4-5. Vector Sum Pipeline	69
4-6. Hough Transform on $n \times n$ Picture	69
4-7. Band Numbers for $\theta = 67.5^\circ$	71
4-8. A Contraction of 16 Processes to 4 Processors	74
4-9. Another Contraction of 16 Processes to 4 Processors	75
A-1. A 15 Node Tree in Poker	95

A-2. Code Names View for Maximum	96
A-3. A Leaf Process	97
A-4. An Internal Process	97
A-5. The Root Process	98
A-6. Port Names View for Maximum	99
A-7. Jacobi Phase 1 Switch Settings	100
A-8. Jacobi Phase 2 Switch Settings	101
A-9. Jacobi Phase 1 Code Names	101
A-10. Jacobi Phase 2 Code Names	102
A-11. Jacobi Phase 1 Port Names	102
A-12. Jacobi Phase 2 Port Names	102
A-13. 6-Cube Interconnection	107
A-14. Batcher's Sort Port Name	108
A-15. DAC Matrix Multiply Port Names	109
A-16. WAP Matrix Multiply Interconnection	112
A-17. WAP Matrix Multiply Code Names	112
A-18. WAP Matrix Multiply Port Names	113
A-19. Tree Folding Interconnection	117
A-20. Tree Folding Code Names	117
A-21. Tree Folding Port Names	118
A-22. Leiserson Layout Interconnection	120
A-23. Leiserson Layout Code Names	120
A-24. Leiserson Layout Port Names	121
A-25. Coalesced Mesh Interconnection	124
A-26. Coalesced Mesh Code Names	125
A-27. Coalesced Mesh Port Names	125

List of Tables

<i>Number</i>	<i>Page</i>
2-1. Timings of the Maximum Algorithm	39
4-1. Contraction Timings for the WAP Matrix Multiply Algorithm	76

Acknowledgements

I wish to thank Larry Snyder for his guidance, understanding, helpfulness, encouragement, and the many hours of discussion. I also thank Larry Ruzzo for many discussions and his careful reading of this dissertation, Jean-Loup Baer for his comments on this dissertation, the many people involved in the BLUE CHiP project, A. Nico Habermann for noticing that Hitech Chess exhibits the CAB paradigm, Carl Ebeling for helping me understand Hitech Chess and Robert Cypher for explaining his Hough Transform algorithm to me.

Parts of this dissertation have appeared before in *The Characteristics of Parallel Algorithms*, MIT Press [66] and in *Proceedings of the 1986 International Conference on Parallel Processing* [65]. This research was supported in part by National Science Foundation grant DCR-8416878 and by the Office of Naval Research Contracts No. N00014-86-K-0264 and No. N00014-85-K-0328.

CHAPTER 1

Parallel Programming

In recent years there has been a great increase in both the interest and availability of parallel computation. Advances in hardware have made it possible to build multiprocessor computing assemblages. As parallel computers become more widespread in research and commercial communities, more programmers will be writing parallel programs.

Many programmers do not have experience in the design of parallel algorithms. Their experience in sequential computation, while valuable, may not provide the tools needed for efficient algorithm design and implementation for a parallel environment. This raises the obvious question: What are the techniques and tools for developing efficient parallel algorithms?

In serial computation there are several recognized programming paradigms, such as the divide-and-conquer, the greedy and the dynamic programming techniques [3]. These computational methods are not algorithms *per se*, but rather they are problem solving strategies that are frequently used in structuring efficient algorithms. Thus, paradigms are the high level methodologies we recognize as common to many of our effective algorithms.

In parallel computation we expect to find similar high level methodologies common to many of the effective parallel algorithms. These parallel programming paradigms, if identified early enough, may help the programmer understand parallel computation and the unfamiliar process of developing parallel algorithms.

This process of parallel programming is further complicated by the diversity in the field of parallel computation. Parallel programming could be writing a program in a data flow language like VAL [61], writing a program for the Cosmic Cube in Cosmic

Cube C [78], writing a program in FORTRAN that is compiled with a parallelizing compiler [6, 51], designing an algorithm for a PRAM [32], or one of many other possibilities. Each of these makes assumptions about how parallelism should be expressed and about the underlying model of parallel computation.

While we expect the parallel programming paradigms to give information about commonalities found in parallel algorithms, we do not expect a single paradigm to be applicable to all models of parallel computation. Thus, for example, the "to do list" paradigm -- keep a queue data structure of task descriptors and have server processes at the completion of a task select a new task from the head of the list -- requires all processors to fetch from the same memory cell (list head) and thus seems to favor a shared memory implementation [37].

In this dissertation, we will limit ourselves to a single model of computation and consider parallel programming paradigms that are useful for that model. In the remainder of Chapter 1, we choose a model of parallel computation and look more into the advantages of studying programming paradigms. In Chapters 2 through 5, we study several paradigms by examining algorithms exhibiting them. In Chapter 6 we conclude with a look at future work available in the area of parallel programming paradigms.

1.1. Parallel Machines and the Type Architecture

There is a plethora of parallel computation models and architectures. Unlike the models of sequential computation, none of the parallel models has emerged as the quintessential model. In the theory community, the paracomputer [76] or PRAM has become preferred over other models such as circuits [17], aggregates [17], and hardware modification machines [27]. There are even variations in the paracomputer type modes, such as Goldschlager's SIMDAG [36], Dekel and Sahni's shared memory machine (SMM) [24] and the multiple variations of the PRAM [29, 32].

Actual hardware designs are just as varied, ranging from the small parallelism shared memory designs such as the Sequent [30] and Lawrence Livermore National Laboratory's S-1 [93] to the non-shared memory message passing machines such as the

CHiP architecture [80], the cube connected cycles [74], the Ultracomputer [76], and the Cosmic Cube [78]. There are also the so-called dance hall architectures [50] where memory is shared by way of a combining network. Examples of the dance hall architecture include the NYU Ultracomputer [37], the PASM Computer [43], the Cedar Computer [33], the Butterfly [19], and the RP3 [70].

With all the architectural diversity, the programmer is required to know exact details about a machine before designing an algorithm. An alternative to this diversity is found in Snyder's Type Architecture concept [82]. A type architecture is an idealized machine that describes salient features and ignores unimportant ones. This is not a rigid specification to which all architectures must conform. In fact, it is not expected that one type architecture will be sufficient for parallel computation.

In reviewing the previous architectures we see that two main styles of architectures arise, shared memory and non-shared memory. The paracomputer is a reasonable type architecture for the shared memory machines. The paracomputer [76] is a machine with N identical processors which share a common memory. All processors can access the memory simultaneously in a single time unit. The most general model allows both simultaneous reads and simultaneous writes to a single memory cell.

While the paracomputer is a reasonable type architecture for shared memory machines, none of the non-shared memory machines is abstract enough to be a good type architecture. For this reason, we accept Snyder's Candidate Type Architecture (CTA) [82] as a reasonable type architecture for non-shared memory machines. The CTA is defined as:

"A finite set of sequential computers connected in a fixed, bounded degree graph with a global controller."

This CTA speaks to realizable parallel architectures in several areas. A real machine will have a finite set of processors. This can be viewed as a resource similar to memory in a sequential machine. These processors are sequential computers that are operating asynchronously. This does not exclude limited parallelism at the processor

like pipelining or I/O co-processors. The communication is through a fixed, bounded degree graph, where messages are passed between processors through some communication channel instead of through common memory. And, finally, there is some kind of a global controller of the processors.

The previously mentioned non-shared memory machines, with the exception of the Cosmic Cube, fit into this type Architecture. The Cosmic Cube fails only in that it is not a bounded degree graph, with $O(\log n)$ connections per node. Even with this deviation, the Cosmic Cube or more generally, the n -cube is still realizable in current technology [78]. As machine sizes grow, this may not be the case.

We now have two type architectures from which to choose. The paracomputer has several drawbacks, of which the most serious is the unit cost assumption [82]. Also, while it is easy to "simulate" a CTA on a paracomputer, it is not easy to "simulate" a paracomputer on a CTA. The methods used to simulate a paracomputer on a CTA are discussed in Chapter 5. All of these methods require at least $O(\log n)$ processing for each "shared memory" reference. Some methods also require synchronous processors and require all processors to cooperate to access the shared memory.

Because algorithms for a CTA are directly applicable to the paracomputer and it is difficult to fit paracomputer algorithms on a CTA, we choose the CTA as our model of parallel computation. Our algorithms and discussion assume a machine conforming to the CTA. Because the CTA does not enforce any particular interconnection structure, we do not assume any particular interconnection structure for algorithms, but let the natural communication needs of the algorithm define the interconnection structure. We deviate in one place from the CTA in that we will consider graphs with $\log n$ degree, specifically the n -cube interconnection structure.

1.2. Programming Paradigms

Having chosen the CTA as our model of parallel computation, we now return to programming paradigms. Remember that programming paradigms are not algorithms *per se*, but are problem solving strategies frequently used in structuring efficient

algorithms. The benefit in identifying paradigms, besides the pleasant and perhaps scientifically important insight that apparently dissimilar problems can be solved by similar means, is the fact that the set of recognized programming paradigms becomes the programmers "tool kit." It is a set of known-to-be-useful strategies that he can use to structure the development of new or improved algorithms. Thus, the programmer not only has a place to begin when developing a new algorithm, but more importantly, experience and knowledge from earlier problem solutions can be directly transferred to a problem by means of the paradigm. As we have noticed, both benefits are important in the parallel computation arena, since to date programmers have generally had little direct experience with parallelism. But for parallel computation in general, and the CTA model specifically, there is an even more compelling reason to concentrate on programming paradigms.

Programming paradigms generally encapsulate information about useful patterns of data reference, which in the case of parallel computation simply says that paradigms generally encapsulate information about useful *communication patterns*. Since providing efficient and effective communication is the problem in parallel computer architecture, the paradigms serve as a specification of which communication patterns are most useful and hence should be supported well.

There is a cautionary remark to be made on precision: it is difficult to define paradigms precisely. We know of no adequate and convenient formulation that can be used. Since paradigms are not algorithms, one does not present them in a programming language, nor are there algorithmic schemata that could be presented in some metalanguage. They can be illustrated by example (which we will do), but this is an inadequate definitional means since it is rarely clear which properties are special and which are general. Still, paradigms are a phenomenon that programmers understand, and in sequential computation, we have learned them by some means. Our approach will be to describe them verbally and augment the description with examples.

Our discussion will center around three paradigms. Chapter 2 introduces the compute-aggregate-broadcast paradigm, focusing on the various ways in which this

paradigm is exhibited. Chapter 3 focuses on how the divide-and-conquer paradigm is used for CTA model. Chapter 4 considers the systolic and pipelining paradigms. In Chapter 5 we briefly look at a fourth paradigm, reduction, and consider the methods for "sharing memory" on a CTA architecture.

1.3. Contraction

There is yet another benefit from studying paradigms. Because the paradigms identify common features of algorithms, it may be possible that these common features can be exploited by general techniques or algorithm transformations, such as techniques for increasing performance. These techniques that exploit common features then are applicable to every algorithm of the paradigm using that common feature. This is the case for the contraction of parallel algorithms.

Most parallel algorithms are designed assuming that processors are an unlimited resource. This assumption manifests itself by the algorithm utilizing a problem size dependent number of processors. The *algorithm contraction problem* arises when an algorithm that is designed for use on n processors must be implemented on a physical parallel computer with only $p < n$ processors. In order to make the algorithm conform to the real machine, several of the algorithm's "processors", now considered processes, are grouped together and executed on a single physical processor. This activity is called *contraction*[11].

How contraction is performed can have a significant effect on performance. Consider two examples based on an $n \times n$ grid of processes, *i.e.* the processes communicate with their four nearest neighbors:

- (1) There is much process-to-process communication and approximately equal computation required of each process.
 - (2) There is little process-to-process communication and the amount of computation per process is proportional to its j index, *e.g.* process i, j iterates j times.
- Suppose we have only one fourth the required number of processors and now compare two ways of forming contractions of four processes per processor[11]: *Coalescing*

groups of adjacent 2×2 subarrays; *folding* groups as if the grid is folded in half and then in half again, *i.e.* i, j ($1 \leq i, j \leq \frac{n}{2}$) is associated with $i, n-j+1$, $n-i+1, j$ and $n-i+1, n-j+1$. Clearly, algorithm (1) should be contracted by coalescing because the process-to-process communication for the processes sharing the same processor will become intraprocessor communications (*i.e.* fast memory references) rather than slow interprocessor communication; folding would not be as attractive because no communication is saved by locality. Alternatively, algorithm (2) should be contracted by folding because the work is balanced since each processor will perform a matching amount of long and short computations; coalescing would not be as attractive because the processors receiving processes with large indexes will become a bottleneck.

There are two methods available for performing contraction. Using the results of Berman and her colleagues[12], an algorithm can be automatically contracted, and this seems to be the best approach when nothing is known about the algorithm. At the other end of the spectrum, the programmer has "complete" knowledge about the algorithm and can perform the contraction manually [65] by deciding which logical processes are to be mapped to the physical processors.

How should the programmer be guided when performing his own contraction? We consider this an important question because contraction is a nontrivial problem for parallel programmers [82]. This nontrivial nature makes the idea of a general contraction technique for a paradigm more attractive. In the remaining part of this section we make a first step toward answering this question by developing tools for reasoning about the merits of different contractions.

Consider an instance of the algorithm contraction problem. We are given an algorithm A that requires n processes. We assume that A conforms to the CTA model. As we saw in the previous examples of contraction, variable amounts of computation for each process can effect how contraction is performed. To remove this factor and due to the fact that many of the algorithms we will study have balanced computation requirements, we assume that the processes of A require equal amounts of computation.

Assume that the physical parallel computer has p processors. We would like the physical processors to have balanced utilization. Since the processes of A have balanced computation requirements, this is done by mapping the same number of processes to each physical processor. The number of processes assigned to two arbitrary physical processors should differ by at most an additive constant c . For most contractions, we would like to have $c=1$.

The contraction induces a communication graph for the p physical processors. This new graph is defined by processes needing to communicate with other processes not mapped to the same physical processor. We assume that if a process in processor i needs to communicate with a process in processor j , there is a physical edge connecting the two processors. We say there is a *logical edge* between the processes using the physical edge. The contraction may map many of these logical edges to one physical edge in the new graph. That is, we are allowing only one edge between physical processors. Logical edges share a physical edge by multiplexing. Because we are using the CTA model, we require that the new graph to be a bounded degree graph.

As an example of contraction, let us assume we have an algorithm with a tree graph as in Figure 1-1. Consider the contraction to 5 processors shown in Figure 1-2. This contraction is produced by mapping sub-trees of size $\frac{n}{p}$ to each physical processor. This caused an increase in the degree, *e.g.* the new root vertex has four descendants. Using this kind of a contraction, mapping sub-trees to processors, it can be shown that

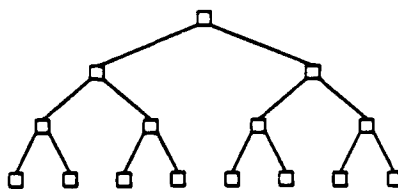


Figure 1-1: A 15 Node Tree

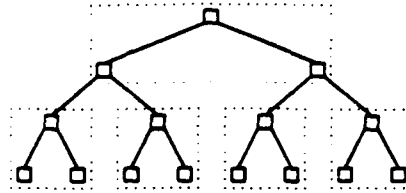


Figure 1-2: A 5 Processor Contraction

given p processors, contracting an algorithm with at least p^2 processes requires degree $p-1$.

Next, consider a contraction of the tree to 4 processors as shown in Figure 1-3. This contraction is performed by placing $\frac{n}{p}$ processes from the same level of the tree into the same processor. An extension of this method yields a binary tree in the p processors where the root processor has only one descendant. Figure 1-4 shows a 31 node tree contracted into 8 processors using this technique. Of these two contractions, only the second contraction conforms to the CTA.

We will be considering the contribution of the communication time to the performance of the contracted algorithm. Unless otherwise stated, we assume that a communication between processing elements costs a fixed time t_c . During this time, no other communication in the same direction may take place. We are specifically allowing all

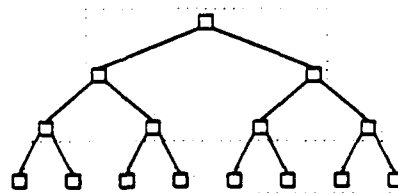


Figure 1-3: A 4 Processor Contraction

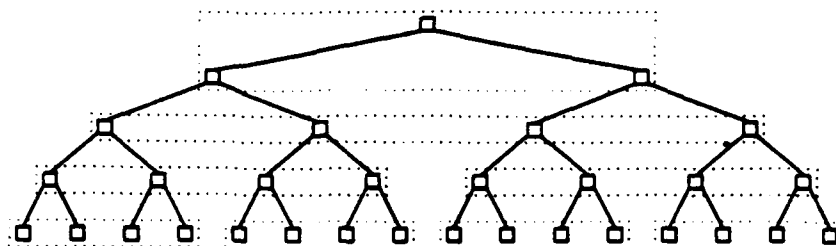


Figure 1-4: An 8 Processor Contraction of a 31 Node Tree

edges to have simultaneous communication. Communication internal to a processor costs the fixed time t_i . We also assume that $t_c > t_i$.

As mentioned before, we would like to develop tools for reasoning about the relative merits of different contractions. This includes their communication costs and their execution times. To aid in this objective we give the following definitions.

Let $A=(V, E)$ be an algorithm where V , sometimes referred to as V_A , is a set of processes (vertices and associated programs) with $|V|=n$ and E is a set of edges (v_1, v_2) , where $v_1, v_2 \in V$.

Let $M(A, p) = B$ be a contraction of algorithm A into algorithm B where B uses p processors and $p < |V_A|$. The contraction M maps elements of V_A onto V_B such that the number of elements of V_A mapped to an arbitrary element of V_B differs by no more one from the number of elements of V_A mapped to any other element of V_B .

Let $w(e)$, the weight of e , for $e = (v_1, v_2)$, be the larger of the number of messages from v_1 to v_2 and the number of messages from v_2 to v_1 .

Let $K(A) = \underset{e}{\text{MAX}} w(e)$, for $e \in E$, be the communication "cost" of A . This cost is an estimate of the minimum communication time required for the algorithm. Due to dependencies, the actual communication cost may be more.

Let $T(A)$ be the execution time for algorithm A .

We would like to be able to say: For a given A , p , M_1 , and M_2 , and $t_c > t_i$, if $K(M_1(A, p)) < K(M_2(A, p))$ then $T(M_1(A, p)) \leq T(M_2(A, p))$. This is based on the

notion that the bottleneck edge will be a lower bound on the time required for the execution of the mapped algorithm. However, due to communication dependencies, this may not be the case.

To see that the communication dependencies have an effect on contraction, consider the algorithm and its contractions shown in Figure 1-5. The algorithm performs the following operations: the center two columns of processes send a single message to the two directly connected columns and then the processor labelled "S" starts a message around the ring of processes and waits for that message to be returned. Each edge has exactly one message sent across it, yielding a cost for the algorithm of $K(A) = 1$.

Consider the contraction in Figure 1-5a. This is performed by mapping all edges used in the first communication to internal edges. There is at most one edge between any two processors in this contraction. Therefore the cost of contraction a is $K(M_a(A, p)) = 1$. Now consider the contraction in Figure 1-5b. This is performed by mapping all edges used in the first communication to external edges such that these edges are connected to two processors. This maps $\frac{n}{2p}$ communication edges into a

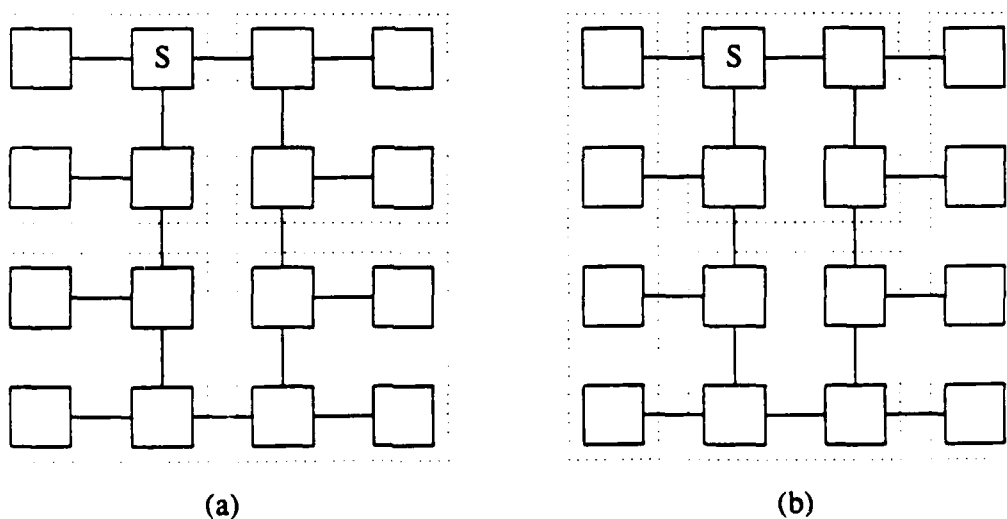


Figure 1-5: An Algorithm with Sequential Dependencies

single physical edge. The cost of contraction b is $K(M_b(A, p)) = \frac{n}{2p}$.

By our previous statement comparing two contractions, we expect M_a to be the better contraction and take less time. The time for contraction a is

$$T(M_a(A, p)) = \frac{n}{2p}t_i + pt_c + \left[\frac{n}{2} - p \right] t_i.$$

The time for contraction b is

$$T(M_b(A, p)) = \frac{n}{2p}t_c + \frac{p}{2}t_c + \left[\frac{n}{2} - \frac{p}{2} \right] t_i.$$

We would like to know if $T(M_a(A, p)) > T(M_b(A, p))$. That gives

$$\frac{n}{2p}t_i + pt_c + \left[\frac{n}{2} - p \right] t_i > \frac{n}{2p}t_c + \frac{p}{2}t_c + \left[\frac{n}{2} - \frac{p}{2} \right] t_i.$$

Simplifying gives

$$\frac{p}{2}t_c - \frac{p}{2}t_i > \frac{n}{2p}t_c - \frac{n}{2p}t_i.$$

$$p^2(t_c - t_i) > n(t_c - t_i),$$

and finally

$$p^2 > n.$$

This can be satisfied for most values of n . That means that the contraction with the larger communication cost runs in less time.

Our communication cost $K(M(A, p))$ is then a lower bound, but not an upper bound on the execution time of the algorithm. The longer running time of contraction a is a direct result of the sequential dependence of the messages. If the algorithm was modified so that all processors in the ring sent their one message in parallel, then contraction a would be the best. It is obvious that this modified algorithm makes much better use of parallelism.

We assume good parallel algorithms do not have this sequential communication dependence. That is, we expect that these contracted algorithms are able to run in time proportional to their communication cost $K(M(A, p))$. The communication costs are then a good indication of execution times. Therefore, we expect the contraction with the smallest communication cost to be the better contraction. That motivates us to map the busiest edges of an algorithm to internal edges.

In the next three chapters, we apply these tools to the contraction of specific algorithms. For the compute-aggregate-broadcast paradigm, we look at the contraction of tree algorithms. For the divide-and-conquer paradigm we consider the contraction of the n -cube. And finally, for the pipelining and systolic paradigms we consider the contraction of mesh based algorithms. Where applicable, we include the results of comparing the contractions by programming them using the Poker parallel programming environment [81].

1.4. Parallel Algorithms

Because our method of presenting these parallel programming paradigms is largely through example, we need a method of expressing parallel algorithms. Often, an informal description of the algorithm is all that is needed. But some algorithms need a more explicit description. This description is via a parallel programming language. As with sequential languages, parallel languages describe a virtual machine. We would like this virtual machine to match our chosen computation model. Not only do we want a method of communicating the algorithms to the reader, but we want a language having an efficient implementation on the same model.

There are several programming languages used for parallel programming from which we could choose. Some compilers for FORTRAN extract the inherent parallelism available in sequential programs. Examples of these systems include Parafrase [51] and PFC [6]. However, there is much convincing evidence that parallel algorithms are different from sequential algorithms. For example, Snyder [82] displays a sequential algorithm which has a high degree of data dependencies and does not admit a high degree of

parallelism, yet he shows that the problem can be solved with a high degree of parallelism if approached from a parallel viewpoint. This leads us to believe that any sequential language can not be sufficient to express the parallelism we would like to see in our algorithms.

Some sequential languages have been extended to include some form of parallelism. Examples of these include FORTRAN 8X [92] and Path Pascal [15]. These languages assume a shared memory parallel environment and implementations of these languages will encounter the problems mentioned in simulating shared memory on a CTA. Although some systems, e.g. Linc- [4], restrict the use of shared memory, there is still the problem of one processor needing to access data in any of the other processors. Even the data flow languages such as VAL [61] treat memory as a global resource.

Communicating sequential processes (CSP) as defined by Hoare [42] provide another class of languages available for parallel programming. These do not provide shared memory, but require sequential processes to communicate by message passing. They do not bound the number of processes with which one process may communicate, which provides the problem of arbitrary communication on a CTA (see Chapter 5). Occam [48] is an example of this class of languages.

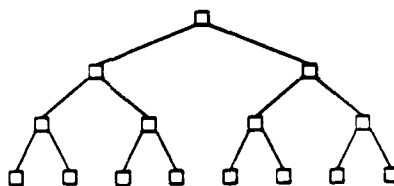
What we want is a language that conforms to the CTA. Snyder refers to these languages as CTA-type languages [82]. The CSP languages and Cosmic Cube C [87] are in principle CTA-type languages. Cosmic Cube C also has the problem of unbounded degree communication, even greater than the $\log n$ Cosmic Cube. This is solved by having message forwarding. If one restricted themselves to communication with a bounded number of other processes, it would conform to all points of the CTA.

A better example of a CTA-type language is Poker [81]. A program consists of a bounded degree graph where the vertices are processors and the edges are communication channels, a set of processes, and an assignment of processes to processors. A complete algorithm is broken into *phases*, each with its own graph, set of processes and process assignment. A global controller runs the phases in the proper order to complete the

computation. Each process is programmed in a sequential programming language with special constructs for communicating with the processes at adjacent processors and for communicating between phases at the same processor.

We choose a Poker-like pseudo language for presenting our algorithms. We define a graph, a set of processes and some implicit assignment of processes to nodes of the graph. In some cases, we define multiple graphs used at different points in the same process. This is usually done to avoid the extra notation to actually show graphs, processes and process assignments for multiple phases. Each unique type of process will be described in a sequential pseudo language.

For communication purposes, a process identifies a directly connected process by a symbolic name called a *port*. The port names are usually descriptive in terms of the



```

Code leaf;
  Ports Parent;
  Parent ← Value;
End;

Code internal;
  Ports Left, Right, Parent;
  Parent ← max(Value, Left, Right);
End;

Code root;
  Ports Left, Right;
  Maximum := max(Value, Left, Right);
End;
  
```

Figure 1-6: Maximum Algorithm

interconnection graph. The construct $name_list \leftarrow expression$ sends the *expression* as a message to each process identified in the *name_list*. Any time a port name appears in an expression, the process waits for the "next" message to be received from the associated process. When the message has arrived, it is used in the expression as the value of the port name. We avoid making a rigid definition to allow flexibility in describing algorithms. As an example, Figure 1-6 shows the description of a maximum algorithm. Each process contains a variable *Value* for which we want the maximum over all processes. This maximum is found in the root process at the completion.

Throughout this dissertation, we are interested in the algorithms, not how to fit an algorithm on a specific machine. Due to this, we present the algorithms in terms of processes not processors. Also, we let the algorithm define the communication graph, instead of making an algorithm use a specific communication graph. The problem of mapping an algorithm to a particular interconnection structure is treated elsewhere by Berman and Snyder [11] and Bokhari [13].

CHAPTER 2

The Compute-Aggregate-Broadcast Paradigm

In this chapter we present the compute-aggregate-broadcast (CAB) paradigm and its variations. There are many algorithms included in the CAB paradigm, coming from both numerical computation and non-numerical computation. We first start with a definition of the CAB paradigm and give a simple example illustrating the basic concepts. We then explore the variations of the paradigm shown in both numerical and non-numerical computation. Finally, we look at a speed-up technique and contraction, both of which have relevance to a subset of the CAB algorithms.

2.1. CAB paradigm and Jacobi Iterations

Compute-aggregate-broadcast (CAB) algorithms are composed of three basic phases: a compute phase which performs a computation, an aggregate phase which combines local data into a few global values, and a broadcast phase which sends global information to each process. The compute phase and its interconnection structure differ widely from algorithm to algorithm. It may be a complete algorithm, like matrix multiplication, or it may be a single primitive operation performed by each process. The aggregate phase is usually a tree-based computation that combines data from the processes, producing a single global value or a small number of values. Minimum and maximum trees are examples of an aggregate phase. The broadcast phase sends global information or a directive based upon it to all processes. This may be a "keep going" message for iterative algorithms or a value necessary for the following compute phase.

A simple example of a CAB algorithm is a parallel implementation of the classical Jacobi iterative method for solving Laplace's equation on a rectangle. An instance of this kind of problem, the electric field problem, is shown in Figure 2-1. The rectangle is represented by n discrete values, V_{ij} , the voltage at the point. The boundary and the

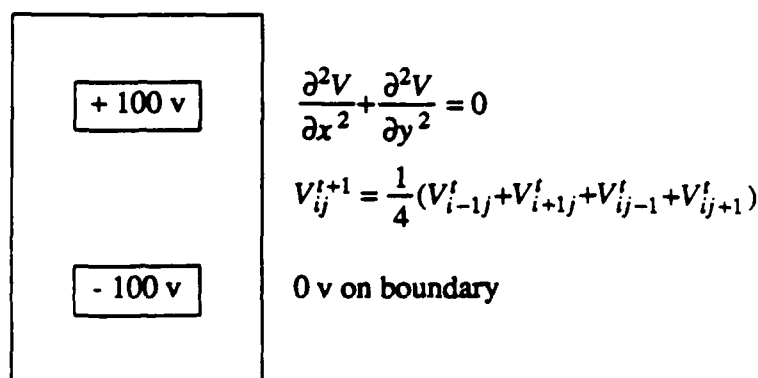
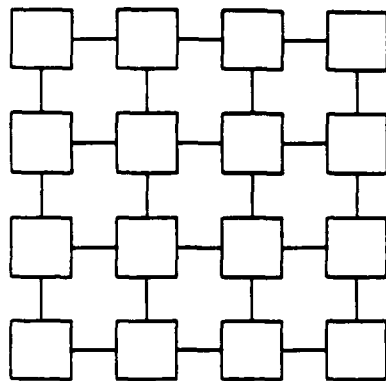


Figure 2-1: Electric Field Problem

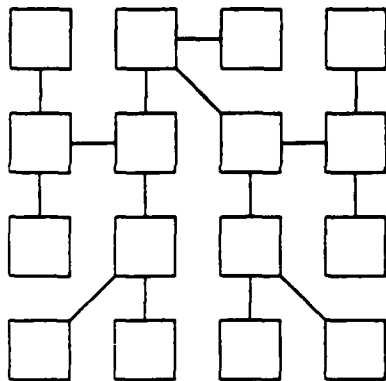
voltage sources are kept at a constant value. An initial guess is computed for each point. Iteratively, a new value, the average of a point's 4 neighbors, is computed for each of the n points. The iteration is terminated when the difference between the new value and the old value at every point is less than some tolerance.

Using n processes, each V_{ij} is assigned to a process. Figure 2-2 shows a 16 process version of the algorithm with the code for an internal node in the aggregate and broadcast trees. The compute phase calculates the new V_{ij} 's and the difference between the old and new value. Using a 4-neighbor mesh interconnection structure (Figure 2-2a), each process exchanges its V_{ij} with its 4 neighbors. It then takes the average of the values received and calculates the difference with the old value. The aggregation phase connects the processes in a binary tree (Figure 2-2b) and computes the maximum of the differences between old and new values. Each process contains a difference value. The leaf processes send their value to their parents. The internal processes take the maximum of their value and their children's, sending the result to their parents. The root process retains the final global maximum. The broadcast phase sends a stop or continue message to all processes depending on whether the global maximum is less than the given tolerance. This broadcast also uses the binary tree.

This algorithm shows the "standard" features of a CAB algorithm. The first phase is the compute phase, using $O(1)$ time to compute a new value and the difference. The



(a) Mesh



(b) Tree

Code Jacobi;

Ports(mesh) North, East, South, West;

Ports(tree) Parent, Left, Right;

V = initial guess;

REPEAT

{ Compute (mesh) }

North, East, South, West \leftarrow V;

NewV = (North+East+South+West) / 4;

Dif = abs(NewV-V);

V = NewV;

{ Aggregate (tree) }

Dif = max(Dif,Left,Right);

Parent \leftarrow Dif;

{ Broadcast (tree) }

done = Parent;

Left, Right \leftarrow done;

UNTIL done;

END Jacobi;

(c) Code

Figure 2-2: 16 Process Jacobi

interconnection is the 4-neighbor mesh. The second phase is the aggregate phase, using $O(\log n)$ time to compute a single maximum. The interconnection is an n node tree. The third phase is the broadcast phase, using $O(\log n)$ time to send a stop or continue message. The interconnection is also an n node tree. The three phases are repeated until the global termination condition is met. The total time for the algorithm is $O(k \log n)$ where k is the number of iterations. The Jacobi iterative algorithm could be described as a $(CAB)^k$ algorithm.

Not all the CAB algorithms are $(CAB)^k$ algorithms. Some algorithms start with aggregate or broadcast phases, although the *sequence* of phases is generally the same with aggregate following compute, broadcast following aggregate, and compute following broadcast. Some algorithms, like the Jacobi, iterate on the phases until some termination condition, like convergence, is reached, while others iterate a fixed number of times. The execution time for a CAB algorithm is $O(k(c+a+b))$, where k is the number of iterations and c , a , and b are the times for the phases. When the aggregate and broadcast phases use trees, the times for a and b are $O(\log p)$ for a p process system.

2.2. Numerical CAB

Having started with the Jacobi algorithm as our first example of a CAB algorithm, we continue looking at numerical CAB algorithms. The SOR algorithm builds upon the simple CAB structure of the Jacobi, showing a more complex compute phase, and the numerical integration algorithm provides the first variation of the CAB paradigm.

2.2.1. SOR

The SOR (successive overrelaxation) iterative method is used to solve linear systems of equations [95]. Adams [1] provides a good discussion about using the SOR on parallel computers. Although we do not discuss the merits of this method or provide the mathematical basis for the algorithm, it is nevertheless a method that yields numerical CAB algorithms. It is very similar to the Jacobi method that we have seen.

We are particularly interested in the multi-color SOR. The common SOR has sequential data dependencies, but the multi-color SOR provides for easier parallel solutions by having different data dependencies [82]. The algorithms follow the same basic outline as the Jacobi. The major differences are found in the actual computation and in the test for convergence.

As an example of the multi-color SOR algorithms, consider the Red/Black SOR using the "five point stencil" [1, 2, 94]. This algorithm can be viewed as having the data

points being computed on a checker board. (See Figure 2-3.) For the compute phase, all points on the red squares are computed using the current values of the black neighbors. Then, all the points on the black squares are computed using the new values for the red points. After all points have been computed, a convergence test is applied to each point. The aggregation is a simple boolean AND tree computing TRUE if all points have converged. The broadcast is a stop or continue message.

If we put a single data point in each process, all the processes containing black points will be idle while the red points are being computed and similarly the red processes will wait while the black points are being computed. By assigning one red and one black point to a process, all processes will be busy all the time. The compute phase now becomes "compute the red point and then compute the black point". The communication structure remains the 4 neighbor mesh. In general, for the multi-color SOR, one data element of each color is mapped into a single process [1]. This may require an 8 neighbor mesh to meet the communication demands. The compute then becomes a sequence of computing different colored points. The aggregate phase now becomes a check to see if all points in a single process have converged and then the standard binary tree to detect complete convergence. The broadcast remains the same.

Black k	Red $k+1$	Black k	Red $k+1$
Red $k+1$	Black k	Red $k+1$	Black k
Black k	Red $k+1$	Black k	Red $k+1$
Red $k+1$	Black k	Red $k+1$	Black k

Figure 2-3: Red/Black SOR

2.2.2. Integration

Our next algorithm, numerical integration [66], exhibits a variation of the CAB paradigm. For numerical integration, we are given the function $f(x)$ and we would like to integrate between a and b using n processes. An easy solution would be to divide up the integration into n separate problems and let each process integrate a small section of the curve. The problem with this technique is that some sections of the curve may need more processing than other sections to get a satisfactory approximation. This provides an unequal load for the processes.

A better solution gives one process control of the work the other processes are performing. Consider the following algorithm.

Integrate $f(x)$ between a and b .

While "not done" do

Control process: decide which section to integrate.

Broadcast: Broadcast the section

Compute: Each process integrates its $1/n$ section of the curve.

Aggregate: Sum the n areas. Return area of section to control process.

End while

The broadcast uses a binary tree. An internal node receives the end points that its subtree is to integrate. It then divides the interval into 3 pieces, a section for the left subtree, a section for the right subtree, and a section for itself, making sure that each process will have the same sized section. The compute uses no communication. Each node integrates its piece of the total section and decides if its value is a sufficiently accurate approximation. The integration can be done by various methods, such as the trapezoid method or Simpson's rule. The aggregation uses the tree to sum the areas computed in the compute phase. If the computed value for a subsection is not sufficiently accurate, the area is not added into the sum and the control process is sent a message identifying

the subsection.

The control process is responsible for choosing the sections to integrate and collecting the final value for the area. The initial step is to broadcast the entire interval to be integrated. The results are a sum of the accurate subsection areas and a list of the subsections needing more processing. This list is assumed to be small. The control process then recursively integrates all of the subsections. This keeps all the processes busy and never recomputes any subsection that has been integrated with sufficient accuracy.

We make two observations about this algorithm. First, this algorithm is a member of the CAB paradigm, but the first phase executed is the broadcast, not the compute phase as in the "standard" CAB. This kind of an algorithm could be called a BCA algorithm. Also, the compute phase did not need any communication and therefore the algorithm needs only a single communication interconnection structure. This variant of the CAB paradigm can be observed in other algorithms, *e.g.* Dekel, Nassimi and Sahni's $O(\log n)$ matrix multiplication algorithm for the binary n -cube[23].

2.3. Non-numerical CAB

We now turn our attention to the non-numerical CAB algorithms. We separate the numerical and non-numerical CAB algorithms for two reasons. First, we find it interesting and somewhat surprising that this paradigm covers both kinds of computation. The CAB paradigm was first found in the numerical algorithms and later discovered in the non-numerical algorithms. The second reason is the diversity of usage of the CAB paradigm in the non-numerical algorithms. The topological sort algorithm uses aggregate and broadcast phases that are not global. It also starts the algorithm with the aggregate phase. Hitech Chess [28] is a hardware implementation of two related CAB subalgorithms, used by a sequential controller. And finally, the CAB paradigm is useful in parallel expert systems based on production systems.

2.3.1. Topological Sort

We first look at the topological sort algorithm. For the topological sort, we are given a directed, acyclic graph $G=(V, E)$. The sort orders the members of V such that if vertex V_i is before vertex V_j in the ordering, there is no path from vertex V_j to vertex V_i in G .

Dekel, Nassimi and Sahni [23] sketch a parallel solution for topological sort using an all-pairs longest paths algorithm. The vertices are sorted using the length of the longest path to the vertex as the sort key. Their algorithm runs in $O(\log^2 n)$ time using n^3 processes. Ruzzo [18] uses the transitive closure of the original graph, sorting the vertices by the number of predecessors in the transitive closure. This also takes $O(\log^2 n)$ time using n^3 processes.

Our parallel solution is achieved in two steps [66]. The first step is to compute a level number, $Level_i$, for every vertex V_i . All vertices with no incoming edges are assigned a level of 0. All other vertices are assigned a level number that is the length of the longest path from any level 0 vertex. Because G is acyclic, the maximum value for a level number is $n-1$. The last step sorts the vertices into an increasing level number order. It is the first step, the computing of level numbers, that can be solved using a CAB algorithm. The sorting step can be done by a suitable sorter.

LEMMA 1: Sorting $\{(Level_i, V_i)\}$ using the $Level_i$'s as the sort key produces a correct topological ordering of the V_i 's.

PROOF: Assume that given a correct assignment of the $Level_i$'s, the final ordering of the V_i 's is not a topological ordering. Therefore, there must be two vertices, V_k and V_l , such that there is a path from V_l to V_k and V_k is ordered before V_l . Since V_k is ordered before V_l , then $Level_k \leq Level_l$. But $Level_l$ is the length of the longest path from a vertex with indegree of zero to V_l . Since, as we assumed, there is a path of length at least one from V_l to V_k , then the longest path from a vertex with indegree of zero to V_k must be at least as long as $Level_l+1$. But $Level_k \leq Level_l$ for V_k to be ordered before V_l , therefore the level assignment is incorrect. \square

Let us now look at the details of the algorithm to assign these level numbers. We assume there are n^2 processes, P_{ij} , where $1 \leq i, j \leq n$ and $n = |V|$. Each P_{ij} contains the corresponding entry, G_{ij} , from the adjacency matrix for graph G . G_{ij} has the value 1 if there is a directed edge from vertex i to vertex j in the graph to be sorted and a 0 otherwise. The state information for each process P_{ij} is contained in three variables, G^k , the current entry in the adjacency matrix, k , the "length" of edges in the adjacency matrix, and $Level$, the local copy of $Level_i$. Throughout the algorithm, the G^k 's define the graph where edges are paths of length exactly k in the original graph.

Let us first consider a useful subalgorithm, `UpdateLevel`. (See Figures 2-4 and 2-5.) In process P_{ij} , `UpdateLevel` takes the state variables G^k , $Level$, and k , and produces a new value for $Level$. The starting value of $Level$ is the length of the longest

```

Code UpdateLevel( $G^k$ ,  $Level$ ,  $k$ );

    /* for processes  $P_{ij}$ ,  $i \neq j$ . */
    Ports RowLeft, RowRight, RowParent,
          ColLeft, ColRight, ColParent;

    /* aggregate on column trees */
    if Column_Leaf then
        if  $G^k = 1$ 
            then ColParent  $\leftarrow k + Level$ 
            else ColParent  $\leftarrow 0$ 
        else /* internal node */
            if  $G_k = 1$ 
                then ColParent  $\leftarrow \max(k+Level, ColLeft, ColRight)$ 
                else ColParent  $\leftarrow \max(ColLeft, ColRight)$ ;

    /* broadcast */
    Level = RowParent;
    if Not Row_Leaf then
        RowLeft, RowRight  $\leftarrow Level$ ;

End UpdateLevel;

```

Figure 2-4: UpdateLevel Algorithm, Non-root Process

```

Code UpdateLevel_Root( $G^k$ , Level, k);

    /* for processes  $P_{ii}$  */
    Ports RowLeft, RowRight, ColLeft, ColRight;

    /* aggregate */
    Level = max(Level, ColLeft, ColRight);

    /* broadcast */
    RowLeft, RowRight  $\leftarrow$  Level;

End UpdateLevel_Root;

```

Figure 2-5: UpdateLevel Algorithm, Root Process

path terminating at vertex V_i of length less than k . The final value of *Level* is the length of the longest path terminating at vertex V_i of length less than $2k$.

The n^2 processes are connected in a variant of the mesh of trees [57]. Each row and each column is connected into a tree with the root located at the process on the diagonal. (See Figure 2-6.) The computation of the new levels is based on the fact that if there is an edge from V_i to V_j in the graph G^k then $Level_j$ must be k greater than the $Level_i$ because there is a path of length exactly k from V_i to V_j in the original graph. At process P_{ij} , if $G^k = 1$, then there is a path of length exactly k from V_i to V_j and the new level of V_j is at least $Level + k$. By taking a maximum of these new level numbers across column j , we now have the new level for V_j . This is done for all columns at the same time. The trees in the columns are used for this aggregation step. To set up for the next UpdateLevel, we must make sure that *Level* in each P_{ij} is updated to the current level of V_i . Because the roots of the trees are on the diagonal, the correct value for the level of V_i is in process P_{ii} . A simple broadcast over the row provides the correct value of *Level* to all processes in row i . This is done using the trees in the rows.

LEMMA 2: Code UpdateLevel, given correct input, correctly computes new level numbers.

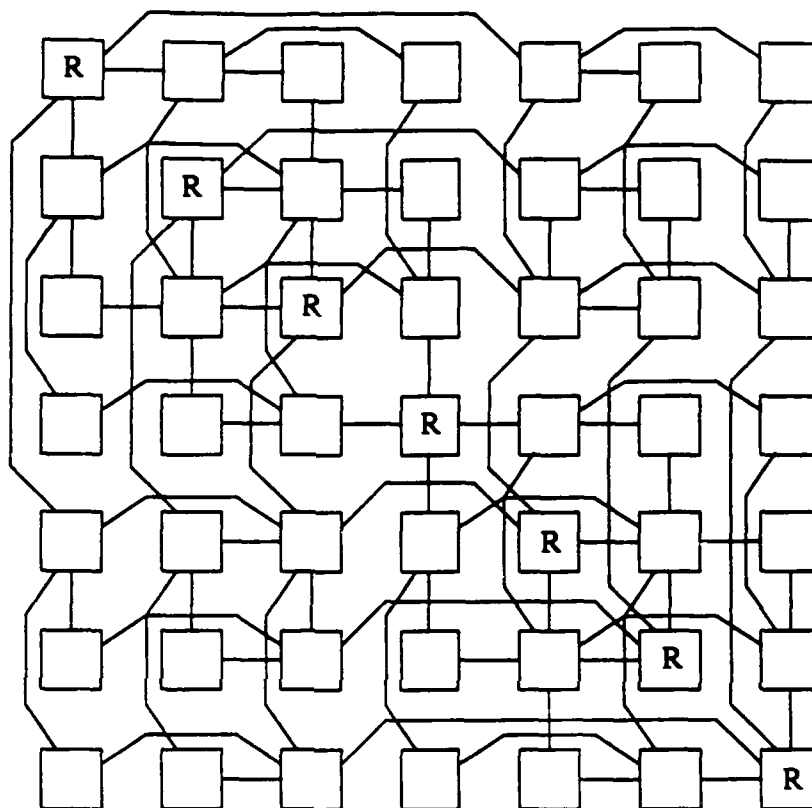


Figure 2-6: The Mesh of Trees with Roots on the Diagonal

PROOF: At every process P_{ij} , $Level = Level_i$, the length of the longest path terminating at vertex V_i of length less than k . Therefore, each column j contains the correct level numbers for all vertices. Consider the values received by the root process P_{jj} from the column sub-trees computing the maximum function. There are 2 cases to consider.

Case 1 is where there are no paths of length exactly k terminating at vertex V_j . In this case, $G^k = 0$ for all processes in column j . The leaves all return the value 0 and the internal processes do not use local values in the maximum. Therefore, both sub-trees return the value of 0 to the root process. The value of $Level$ then remains the same. (Note: The leaves must return 0, not $Level$. If they did return $Level$ the value of the maximum would be $\max_i Level_i$.)

Case 2 is where there is at least one path of length exactly k terminating at vertex V_j . In this case, at least one $G^k = 1$. Assume process P_{ij} has $G^k = 1$. There is then a path of length k from V_i to V_j . The level of V_j must then be $k + Level_i$. Since at P_{ij} , $Level = Level_i$, the value $k + Level$ is used in the maximum computation. The maximum function computes the largest such value in the column j and assigns it to $Level_j$. This new value is between k and $2k$ and therefore must be the length of the longest path terminating at vertex V_j of length less than $2k$.

The final broadcast sends the new level number to all processes so that at process P_{ij} , $Level = Level_j$. \square

Now that we have the UpdateLevel subalgorithm, we can look at the complete topological sort algorithm. (See Figure 2-7.) The initial state, $G^k = G_{ij}$, $Level = 0$ and $k=1$, sets up the state variables for the first call to UpdateLevel. That is, G^k is the initial graph, all edges are paths of exactly length 1 and $Level < 1$. After the first call to UpdateLevel, only vertices with no inedges still have a level of zero. All other vertices have level of one because a path of length 1 terminates at them. The first part of the loop prepares the state variables for the next call to UpdateLevel by squaring G^k and doubling k . The call to UpdateLevel then adds in paths of length exactly k into $Level$. After $\log n - 1$ squarings of G^k and calls to UpdateLevel, each $Level_i$ has been set to the correct value. The final step of the algorithm sorts the vertices by their level numbers.

LEMMA 3: Code Topological_Sort computes correct level numbers.

PROOF: The original state has the original graph in G^k and all level numbers at 0 and $k = 1$. This is correct input for UpdateLevel. By Lemma 2, UpdateLevel correctly computes level numbers less than $2k$. Therefore, after the call to UpdateLevel, all level numbers are less than 2. By squaring G^k and doubling k , we now have correct input for UpdateLevel. Again, by Lemma 2, the level numbers are computed correctly. After each call to UpdateLevel, we must again square G^k and double k . After the last call to UpdateLevel, $k = \frac{n}{2}$ and all level numbers are less than $2k$ or n . Since the longest path

Code Topological_Sort;

```
/* for process  $P_{ij}$  */
```

```
 $G^k = G_{ij}$  in original graph;
```

```
Level = 0;
```

```
k = 1;
```

```
UpdateLevel( $G^k$ , Level, k);
```

```
For index = 1 to  $\log n - 1$  do
```

```
     $G^k = \text{BooleanMatrixMultiplication}(G^k, G^k);$ 
```

```
    k = 2k;
```

```
    UpdateLevel( $G^k$ , Level, k);
```

```
End for;
```

```
Sort(  $\{V_i\}$ ,  $\text{Level}_i$  is the key for  $V_i$ );
```

End Topological_Sort;

Figure 2-7: Topological Sort Algorithm

in G is $n-1$ and Level_i is the length of the longest paths terminating at vertex V_i of length less than n , Level_i is the length of the longest path from a level 0 vertex to vertex V_i . \square

The topological sort algorithm takes $O(n \log n)$ time with n^2 processes. This is because the boolean matrix multiply takes $O(n)$ time with n^2 processes and there are $\log n$ iterations. Due to the trees in the UpdateLevel, the AB phases run in $O(\log n)$ time and this is not the dominant factor. This is not the best algorithm for n^2 processors. Ruzzo's algorithm takes $O(n)$ by using the $O(n)$ transitive closure algorithm given by Guibas, Kung and Thompson [38]. If n^3 processes are used, the matrix multiply time is reduced to $O(\log n)$ [23, 64, 74], yielding $O(\log^2 n + s)$ time for the algorithm, where s is the time for sort algorithm. Notice that the matrix multiply and sorting algorithms may require a different communication structures than the UpdateLevel algorithm uses. It is possible to use an n-cube to compute UpdateLevel in time $O(\log n)$, making it possible to use a single interconnection structure for the complete algorithm by using the matrix multiplication algorithm presented in Chapter 3 and Batcher's sort [10].

We make a few observations about this algorithm. First, as we said before, the aggregation and broadcast are not global, but localized to the rows and columns. Also, notice that the aggregate and broadcast phases are done first, before any computing. If we include the sort as the final compute phase, this algorithm is really an ABC (aggregate-broadcast-compute) algorithm. Finally, the compute phase is a complex phase that may be solved using a different paradigm, *e.g.* pipelining or divide and conquer.

2.3.2. Hitech Chess

Our second example of non-num. CAB is Ebeling's Hitech Chess machine [28]. Hitech Chess is a specially designed machine for playing chess. It is an architecture using parallelism for move generation and position evaluation which has been implemented using custom VLSI. By March 1986, it had the USCF rating of 2352, 150 better than the previous best computer chess system.

The machine can be viewed as a global controller, a parallel move generator and a parallel position evaluator. For each move of the game, the controller performs an α - β search, using special purpose hardware to increase the performance of the search. The machine searches 200,000 positions per second. It is the parallel move generator and parallel position evaluator that exhibit the CAB behavior.

Figure 2-8 shows the logical diagram of the Hitech Chess machine. The compute phase is the same for the move generation and the position evaluation. Given the state of the chess board, they compute every legal move from that state. This is done by having a processor for every possible move that could be made on the chess board. There are about 4000 "ever-possible" moves for each side, giving about 8000 moves to check for legality. The aggregate phases take the legal moves as input and produce a single output.

The aggregate phase for the move generator produces a single move. Given a board position, there is a set of legal moves. These moves are ranked for order of inspection for the α - β search. The controller asks the move generator for the i^{th} move

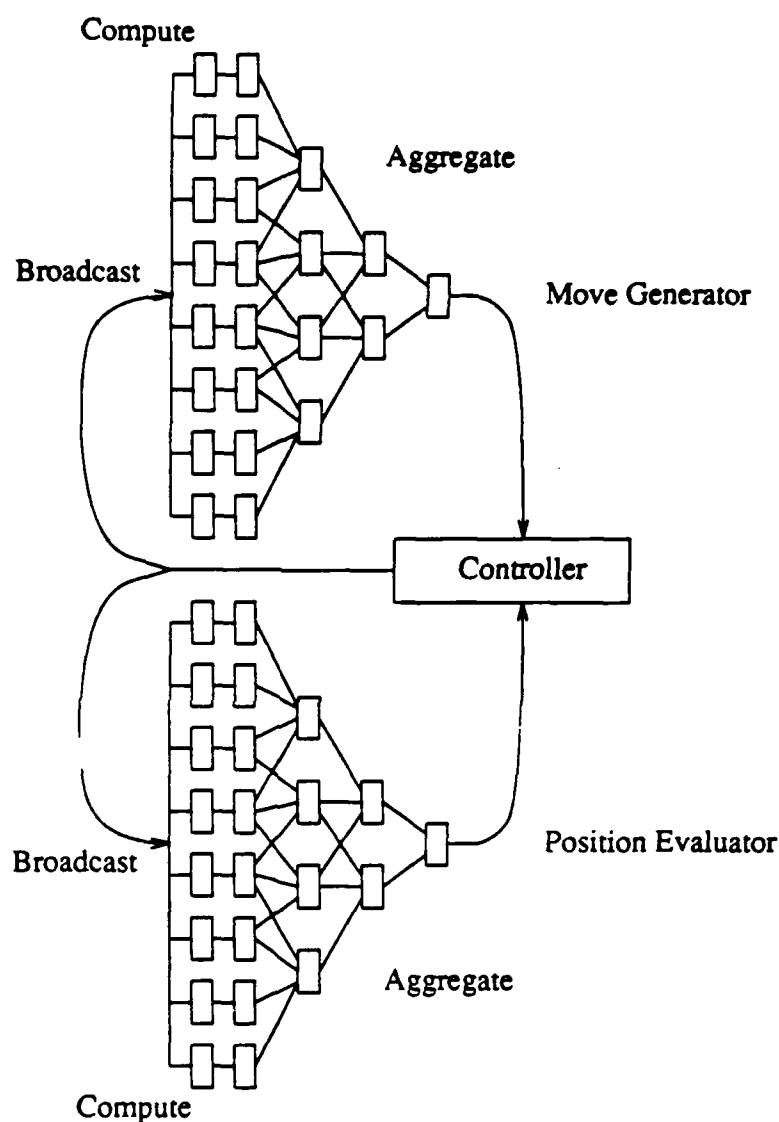


Figure 2-8: CAB in Hitech Chess

in the ranking. For example, the first time the search sees a board position, it wants to search down the most promising subtree. The controller would then ask for the best or first move from the move generator. When the controller has backed up to that board position, it has already evaluated the first move and it now wants the second best move. Since the moves from that position were not saved, the move generator recomputes all the legal moves (compute phase) and selects the second best move as the output of the

aggregation phase.

The aggregate phase for the position evaluator produces a number that estimates the worth of that position. The controller then takes the next move from the move generator and the value of the current position and decides whether to search the subtree produced by the given move or to back up the search tree. This decision is broadcast to both the move generator and the position evaluator. They update their copies of the state and then start the compute phase for the new board position.

There several points of interest in Hitech Chess in relation to the CAB paradigm. First is that while the aggregation interconnection is not a true tree, it is still logarithmic in depth. Also, the actual aggregation step is modified in operation by both the position in the search tree and, specifically for the position evaluation, by the place in the game. And finally, two CAB algorithms are used to produce a single broadcast.

2.3.3. Production Based Expert Systems

As our last example of non-numerical CAB we look at expert systems, more specifically, rule based production systems. It is believed that parallel processing is needed to achieve the goals set for these systems [25]. While it is not within the scope of this dissertation to discuss in detail the use of parallelism in expert systems, we want to show that current algorithms employ the CAB techniques.

In general, production systems [31, 40, 67] are composed of a set of rules, called production memory (PM) and a set of assertions, called working memory (WM). Each rule of the PM is composed of a left hand side, used to match assertions in the WM, and a right hand side, used to specify operations to be applied to the WM. The operation of the production system is a three step process. The first step is a match phase, where left hand side of the rules in the PM are compared to assertions in the WM. Each rule that matches is added to a conflict set. The second step is to select one rule in the conflict set for execution by the the third step. The action of the third step changes the WM. These three steps are repeated until some goal condition is met.

DADO is a parallel processor designed to execute production systems [84]. The processors are connected in a tree. Figure 2-9 shows the structure of the DADO machine. Each rule in the PM is assigned to a separate subtree. The subtree contains copies of relevant parts of the WM for the subtree's rule. The DADO algorithm, in its simplest form, operates in the following way:

1. Each PM node and its subtree determine if the rule associated with the subtree is part of the conflict set.
2. The tree above the PM level takes the conflict set from the PM nodes and reduces it down to a single rule to be applied to the WM.
3. The selected rule is broadcast to all PM subtrees. The PM subtrees update their copy of WM.

It is easy to see that step 1 is a compute step, step 2 is an aggregate step and step 3 is a broadcast step. The three steps are repeated until the goal condition is met, giving rise to a standard $(CAB)^k$ algorithm. The tree structure of CAB algorithms is the interconnection structure of the entire machine. With fixed size subtrees for each rule, the depth of the tree is logarithmic in the number of rules, the expected bound for CAB algorithms.

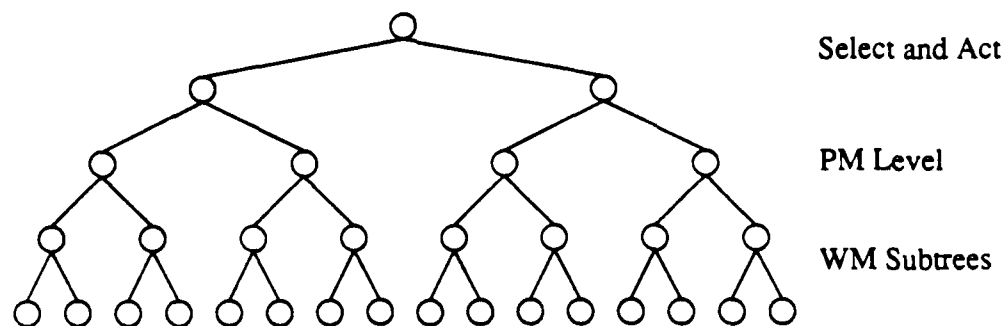


Figure 2-9: DADO Processor Structure

Gupta describes implementing the OPS5 production system on the DADO [39] with three different algorithms. All of them have the same $(CAB)^k$ basic structure. The differences are found in the compute step and in the information used in the aggregate and broadcast steps.

2.4. A Method for Improving Speed

One of the reasons for studying paradigms is the applicability of a technique to several algorithms using the same paradigm. One such technique for CAB algorithms is a method for improving speed. The CAB algorithms which can use this method have the following characteristics:

- a) run time is dominated by the aggregate and broadcast phases, compute times less than $O(\log n)$, often times $O(1)$,
- b) consecutive compute phases do not require information from the aggregate and broadcast phases between the compute phases, and
- c) extra computation phases will not result in invalid results.

The speedup method changes the algorithm to balance the time in the compute phases with the time in the aggregate and broadcast phases. This is accomplished by repeating the compute phase several times before a single aggregate-broadcast phase. An algorithm originally described as $(CAB)^k$ now becomes $(C^l AB)^m$, where $m = \left\lceil \frac{k}{l} \right\rceil$.

Assuming that the time for C^l is less than or equal to $\log n$, the time for an aggregate-broadcast phase, the total time for the algorithm is $O(m \log n)$ instead of the original time of $O(k \log n)$.

For the Jacobi iteration algorithm, the compute time is $O(1)$, allowing $l = \log n$, yielding total time of $O(\log n)$ if k is smaller than $O(\log n)$ or $O(k)$ if k is larger than $O(\log n)$. This method also works for the SOR algorithms. The numerical integration algorithm can make use of this by spending $O(\log n)$ time integrating its own section before reporting on the area or the lack of sufficient accuracy.

There are CAB algorithms for which this technique will not work. The conjugate gradient algorithm [34] computes an inner product during the aggregate phase and the result of the inner product is required for the next compute phase. It cannot use this technique to decrease execution time. All of the non-numerical algorithms we have seen had either long compute phases, like the topological sort, or they needed the results of the aggregate and broadcast phases before the following compute phase and thus do not benefit from this technique.

2.5. Contraction of CAB Algorithms

Another technique that has applicability to CAB algorithms is contraction. Due to the fact that most CAB algorithms have a tree interconnection structure as part of their aggregate and broadcast phases, we look at how to contract trees, specifically binary trees. These results are applicable not only to the CAB algorithms but to all algorithms that employ the binary tree interconnection structure. Also, since some CAB algorithms have compute phases using other than tree interconnection structures, we look at the problem of multi-phase contraction.

2.5.1. Tree Contraction

As representative of the binary tree algorithms, we choose to study the contraction of the maximum algorithm. Each process in the tree contains a value. Leaf processes send their value to their parents. Internal processes take the maximum of their own value and their children's values and then send the result to their parents. The global maximum will be computed at the root process in $O(\log n)$ time. As stated before, we are focusing on the communication for evaluating contractions. The communication in the maximum algorithm requires one message over each edge. Therefore $K(\text{maximum}) = 1$, the communication "cost" of the maximum algorithm.

We are going to compare the contractions shown in Figure 2-10. In Section 1.3 we saw that we need contractions that yield bounded degree graphs and that the contraction in Figure 2-10a, when extended to p processors, yields a binary tree.

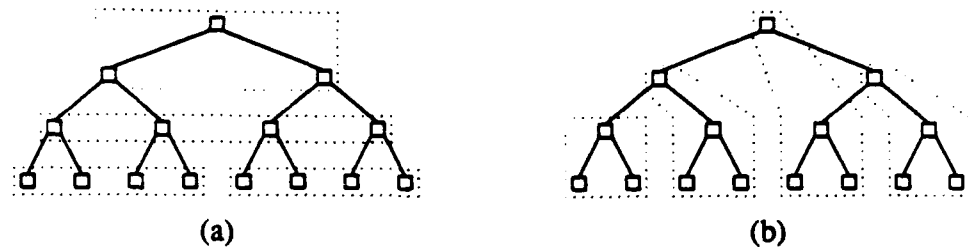


Figure 2-10: Tree Contractions

The contraction in Figure 2-10b yields another bounded degree graph. This contraction is derived by the recursive tree construction given by Leiserson[58]. Leiserson's method of tree construction is shown in Figure 2-11. Given two instances

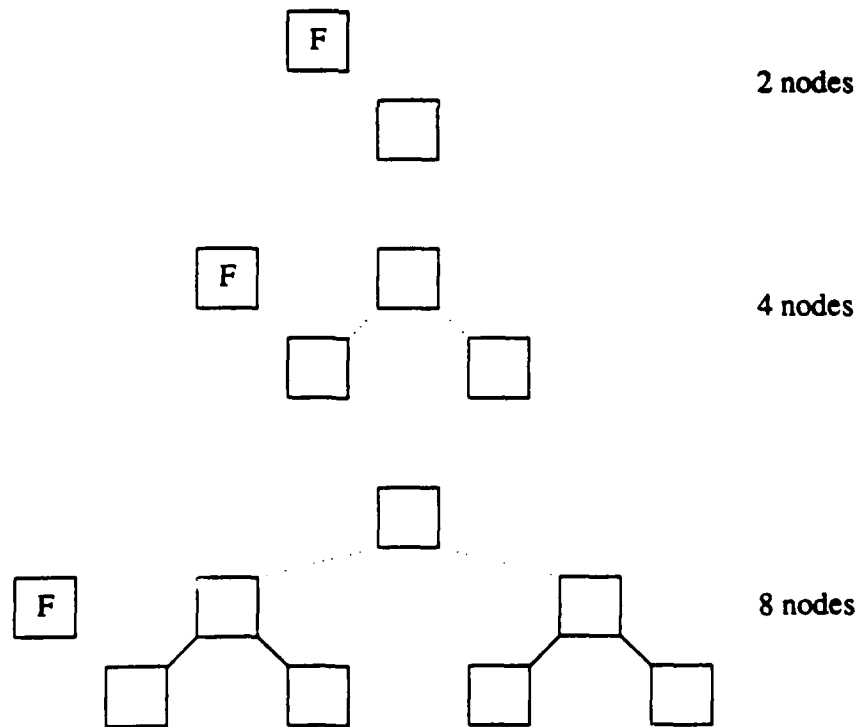


Figure 2-11: Leiserson Tree Construction

of a tree each with an associated free node, shown by the label "F" in Figure 2-11, we can build a new tree and an associated free node. The new edges are shown by dotted lines. This produces a linear area layout in the plane with several desirable properties. The contraction is performed by placing a sub-tree and its associated free node in a single processor. This contraction produces a new graph where nodes have degree of at most 3. The new graph is not a tree.

Consider the contraction in Figure 2-10a. Let us call this contraction $M_1(\text{maximum } p)$, the contraction of the *maximum* algorithm to p processors. Each edge in the original algorithm requires one message. Each edge in the smaller graph contains 4 edges from the original graph. Since we have only one connection between the physical processors, we have 4 messages for each edge. For an arbitrary n (size of original algorithm) and p (the number of processors) we have $K(M_1(\text{maximum } p)) = \frac{n}{p}$.

A similar contraction to Figure 2-10a is touched on by Berman and Snyder[11]. Figure 2-12 shows this contraction. This is achieved by "folding" the tree. As Berman and Snyder notice, this contraction, M_2 , has $K(M_2(\text{maximum } p)) = \frac{n}{p}$. This is also the same contraction shown by Bailey and Cuny as the result of folding their graph embeddings [9].

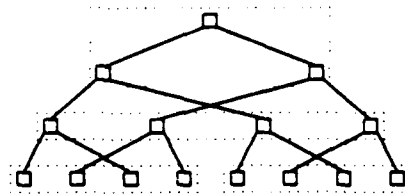


Figure 2-12: Berman and Snyder Tree Contraction

Consider the contraction in Figure 2-10b. Let us call this contraction $M_3(\text{maximum } p)$. We note that each edge in the smaller graph has at most one edge from the original graph in each direction. For an arbitrary n and p we have $K(M_3(\text{maximum } p)) = 1$.

Our experience from Chapter 1 tells us that since M_3 has a smaller K , it is the preferable contraction. Both M_1 and M_2 depend on n and p for their cost. But, M_3 has a constant cost, regardless of n and p . In fact, this contraction is optimum for all tree algorithms that have identical edge weights and unidirectional communication (all toward the root or all toward the leaves).

LEMMA 4: For complete binary tree algorithms with equal edge weights, and unidirectional communication, algorithm contraction based on Leiserson's binary tree layout technique yields an optimum communication cost measure.

PROOF: We first look for a lower bound. Since the tree is connected, the physical processors must be connected. This requires at least one incident edge for each physical processor. The smallest cost $K(M(A, p))$ would be where a maximum of one logical edge was mapped to a physical edge. Therefore, $K(M(A, p)) \geq K(A)$, the cost of the original algorithm.

For the mapping $M_3(A, p)$, each processor contains a complete subtree and an "extra" node. The extra nodes are used in the tree above the subtrees contained in the processors. Therefore, there at most 4 external connections. Of these four, two edges are used to receive(send) data from(to) the children of the extra node, and two edges are used to send(receive) data to(from) the subtree's and the extra node's parents. Since the root of the subtree and the extra node are not at the same level in the tree, edges with data flowing in the same direction can not be connected to the same physical processor. (It is possible to have two of these edges over the same physical edge, but the data moves in opposite directions.) This gives the same weight to the physical edges as the original edges. Therefore, $K(M_3(A, p)) = K(A)$, which is the lower bound. \square

Notice that this layout technique will place two logical edges in the same physical edge for some physical edge. For tree algorithms with bidirectional communication, we then get $K(M_3(A, p)) = 2K(A)$.

To demonstrate these results, the maximum algorithm was programmed using the Poker parallel programming environment [81]. Both M_1 and M_3 were programmed. (The program for 64 data elements on 16 processors is included in Appendix A.) Each contraction was timed using 4 and 16 data items per processor with 4 and 16 processors. The results of these timings are given in Table 2-1. The first two entries for each mapping are the timings for 4 data items per processor. The last two entries are the timings for 16 data items per processors. These numbers are the average time of 5 runs, each run on different data. Both mappings were given identical data for each of the 5 runs. Each "tick" represents a microsecond on the 64 processor Pringle. Integer assignment on the pringle takes about 25 ticks and sending an integer to an adjacent processor takes about 300 ticks assuming both processors are ready at the same time to complete the communication.

This data shows clear superiority for M_3 . With 4 data items per processor it was 2.6 times faster for both machine sizes. With 16 data items per processor it was at least 5.2 times faster. Although these few number of data points do not establish constants of proportionality there are a few points of interest. First, consider the increase in time by

Table 2-1: Timings of the Maximum Algorithm

Maximum: ticks for n (items) on p (processors)				
Contraction	16 on 4	64 on 16	64 on 4	256 on 16
M_1	11484	19945	53341	63536
M_3	4307	7627	8838	12186

changing the data size and machine size by a factor of 4. The increase in time for M_3 is 3320 for 4 data items per processor and 3348 for 16 data items per processor. The difference between these increases is a small 28 ticks. But for M_1 these increase is 8461 and 12195 respectively, with a difference of 3734 ticks. Also, consider the time for 64 data elements on two different machine sizes. The time for M_3 increases by a factor of only 1.2 on the smaller machine while M_1 increases by a factor of 2.7. And finally, it is interesting the effect of increasing the data by a factor of 4 and using the same machine size. For M_3 the times increased by the factors 2.1 and 1.6 for 4 and 16 processors respectively. For M_1 the times increased by the factors of 4.6 and 3.1 respectively.

2.5.2. Multi-phase Contraction

The previous results on tree contraction are useful when an algorithm has only the tree interconnection structure, for example, the CAB numerical integration algorithm. Many of the CAB algorithms, for example, the Jacobi iterations, have compute phases using a different interconnection structure. It is not obvious how to contract multiple phases. For CAB algorithms, we usually have only two phases, a tree interconnection structure and some other interconnection structure. More generally, it is difficult to contract algorithms consisting of multiple phases where each phase has a different interconnection structure.

There are several approaches that can be taken for multi-phase contraction. Each of them assumes that different things are important and may be useful in different algorithms.

In the first approach, each phase of the algorithm is contracted separately, yielding the best results for each phase. Each phase may then have a different mapping from logical processes to physical processors. Therefore, data may need to be moved between processors due to different mappings. This adds data movement phases between the original phases. Because of these data movement phases, this is reasonable only for algorithms which save more by the individual contractions than it costs for the added phases.

In the second approach, the entire algorithm is considered a single phase. The cost of the algorithm is calculated using the union of all phases' interconnection structures and weighted for the number of times a phase is executed. The resulting contraction makes a single assignment of logical processes to physical processes. No extra data movement phases are required. It is possible that every phase is not contracted optimally. It still may be less costly this way than adding the extra phases as in the first approach.

In the third approach, one phase is chosen as the "primary" phase. Contraction is performed for the best results of the "primary" phase. The other phases must use the same process to processor assignment as the "primary" phase. No extra data movement phases are required. Sometimes, the secondary phases are allowed to change their algorithm to match the process to processor assignments.

Because CAB algorithms usually have commutative operations in the aggregate and broadcast phases, the third approach is better. The algorithm is contracted to give the best results for the compute phase. For the aggregate and broadcast phases, the processors are connected up into a tree. Each processor then computes locally yielding a single value from each processor to be used in the aggregate phase. The broadcast needs only to send each processor the broadcast information, not each logical process. This is the method used in published algorithms for the Jacobi and SOR algorithms [1].

CHAPTER 3

The Divide-and-Conquer Paradigm

Divide-and-conquer is a well known paradigm in sequential algorithm development [3,46,77]. A problem is divided up into two or more smaller problems, called sub-problems. Each of these sub-problems are solved independently and their results are combined to give the final result. These sub-problems are just smaller instances of the original problem, giving rise to a recursive solution. Processing may be required to divide the original problem or to combine the results of the sub-problems.

In sequential computation, the sub-problems are solved serially. After dividing up the original problem, each sub-problem is completely solved before starting to solve the next sub-problem. After all the sub-problems have been solved, the results are combined to yield the final solution. A tree structure is often a result of sequential divide-and-conquer. For example, the quicksort algorithm [41] produces a tree of sub-problems.

In parallel divide-and-conquer, the sub-problems can be solved at the same time, given sufficient parallelism. The splitting and recombining process also makes use of parallelism. Because the data is distributed in the processes, these splitting and recombining operations may require process to process communication. Because the sub-problems are independent, no communication is necessary between processes working on different sub-problems.

In this chapter we look at several divide-and-conquer algorithms, showing how this paradigm is used in parallel algorithms. We also investigate the relationship between parallel divide-and-conquer and the binary n -cube. And finally, we consider contraction of divide-and-conquer algorithms.

3.1. Batcher's Bitonic Sort

As the first example of this paradigm we look at Batcher's bitonic merge sort [10]. The problem is that of sorting n data items using n processes. We assume that $n = 2^k$ for some k , and that each process initially contains one data item. The sort should leave the smallest data item in the process labelled 1, the next smallest data item in the process labelled 2, and so forth.

This algorithm contains two instances of divide-and-conquer. Figure 3-1 shows the data exchanges required by the algorithm. The horizontal lines represent a single process and the arrows point to the process which keeps the larger data element. The first application of divide-and-conquer is shown by the box labelled *a* in Figure 3-1. The data (processes) is divided in half. The first half is sorted increasing and the second half is sorted decreasing. When this is completed, the data is in a bitonic sequence. Notice that no processing is required to divide the data.

To combine the results of the sub-problems, the algorithm must turn a bitonic sequence into a completely sorted sequence. The solution to this problem is the second

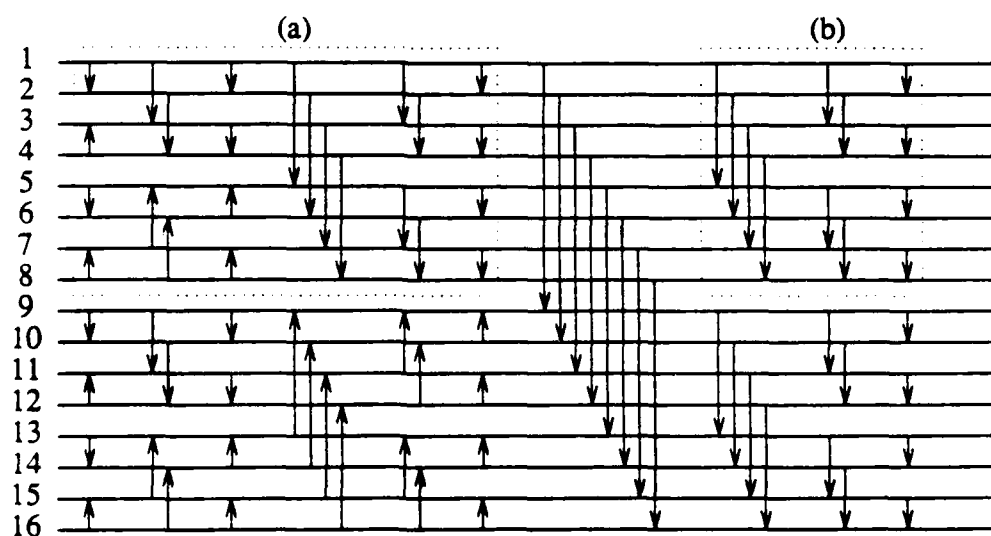


Figure 3-1: Bitonic Sort Data Exchanges

application of divide-and-conquer. To "sort" the bitonic sequence, the algorithm divides the original bitonic sequence into two bitonic sequences such that the first bitonic sequence contains the data for the first half of the sorted sequence and the second bitonic sequence contains the data for the last half. The same process is then used to sort the smaller bitonic sequences as shown by the box labelled *b* in Figure 3-1. Notice that for this application of divide-and-conquer, the splitting required a comparison between corresponding elements of each half. Also, this comparison is the only communication between the halves in the entire sort algorithm.

The full communication structure for this algorithm is the binary n -cube. This can be easily seen by projecting the horizontal lines in Figure 3-1 to a point and retaining the vertical edges. It is also easily seen that the bitonic merge sort takes $O(\log^2 n)$ time. This is due to the second application of divide-and-conquer taking $O(\log n)$ time and being used in each of the $\log n$ sorts.

Notice that this algorithm is not a parallelization of an optimal sequential divide-and-conquer sorting algorithm. If this parallel algorithm were to be run on a single processor, the run time would be $O(n \log^2 n)$. In fact, it is not an optimal parallel sorting algorithm because there are $O(\log n)$ parallel sorts [5]. However, the $O(\log n)$ parallel sorts have very large constants making Batcher's sort the best practical algorithm for reasonable size sorting problems.

3.2. Matrix Multiplication

Our next divide-and-conquer algorithm is the multiplication of two dense $n \times n$ matrices,

$$AB = C$$

using n^2 processes, and assuming $n = 2^k$ for some constant k [64]. (A version of this algorithm for shared memory machines was sketched by Horowitz and Zorat [45].) The processes are viewed as an $n \times n$ array where the processes are labelled PE_{ij} for $1 \leq i, j \leq n$. The matrices A and B are initially distributed in the n^2 processes such that a_{ij}

and b_{ij} are contained in PE_{ij} . After the product we have c_{ij} contained in PE_{ij} .

To begin with, consider the 2×2 case. PE_{11} contains a_{11} and b_{11} . To compute c_{11} the values a_{12} and b_{21} are needed. Similarly, all other processes need only 2 elements not already stored at that process. To provide for direct communication, a grid interconnection structure is used. The processes then send their a_{ij} value to the other process in the same row, and their b_{ij} value to the other process in the same column as shown in Figure 3-2. After this communication, each process, PE_{ij} , has all the data required to compute c_{ij} .

Now consider the $n \times n$ case. The algorithm uses Strassen's [86] matrix decomposition where two $n \times n$ matrices can be viewed as two 2×2 matrices of $\frac{n}{2} \times \frac{n}{2}$ matrices. The 2×2 matrices are then multiplied using matrix product and matrix addition on $\frac{n}{2} \times \frac{n}{2}$ matrices.

Let A_{11} be the upper left $\frac{n}{2} \times \frac{n}{2}$ submatrix of A . Similarly define the other 3 sub-

$$\begin{bmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{bmatrix} \begin{bmatrix} b_{11} & b_{12} \\ b_{21} & b_{22} \end{bmatrix} = \begin{bmatrix} c_{11} & c_{12} \\ c_{21} & c_{22} \end{bmatrix} = \begin{bmatrix} a_{11}b_{11}+a_{12}b_{21} & a_{11}b_{12}+a_{12}b_{22} \\ a_{21}b_{11}+a_{22}b_{21} & a_{21}b_{12}+a_{22}b_{22} \end{bmatrix}$$

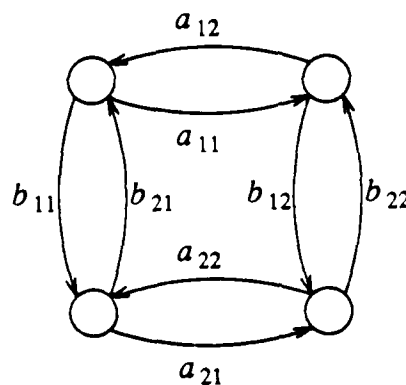


Figure 3-2: 2×2 Product and Communication Structure

matrices, A_{12} , A_{21} , and A_{22} , the submatrices of B , the submatrices of C , and the subarrays of the processes, P . Then A_{ij} and B_{ij} are contained in P_{ij} , $1 \leq i, j \leq 2$. As in the 2×2 case, A_{12} and B_{21} are required to compute C_{11} . If the corresponding processes in P_{11} and P_{12} are directly connected (see Figure 3-3), A_{12} can be sent to P_{11} in parallel with one communication step. B_{21} can be sent to P_{11} using a similar connection scheme in one communication step. The full connection structure connects PE_{ij} with both $PE_{i \pm \frac{n}{2} j}$ and $PE_{ij \pm \frac{n}{2}}$. With A_{11} , B_{11} , A_{12} , and B_{12} in P_{11} , C_{11} can be computed by doing two $\frac{n}{2} \times \frac{n}{2}$ matrix products and one matrix addition. Analogous products can be done using this same algorithm on the other $\frac{n}{2} \times \frac{n}{2}$ matrices. The recursion will stop

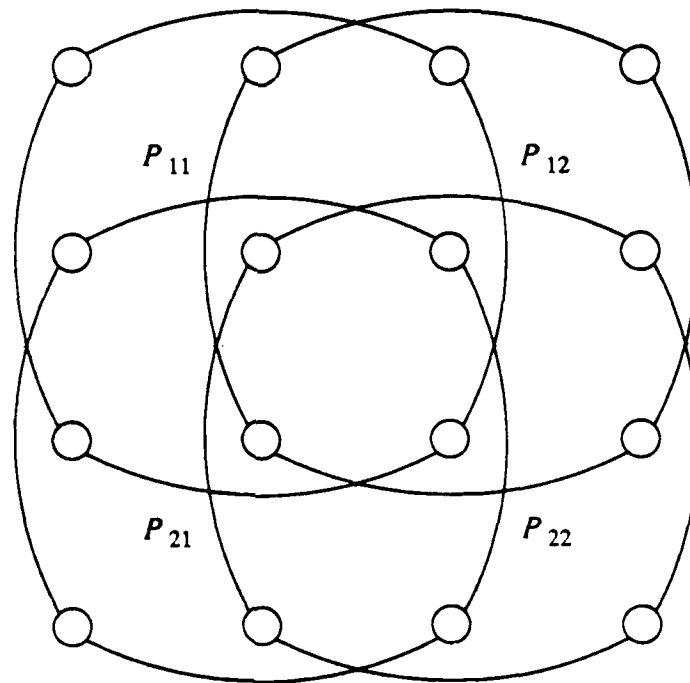


Figure 3-3: 4x4 Connections

after $k-1$ levels when a 2×2 matrix product is done. The matrix addition is performed element by element.

Each recursion level requires it's own interconnection structure for dividing up the problem. The complete interconnection structure, supplying an edge for every communication in the algorithm on every level of recursion, is the n -cube.

The time required for this algorithm is $O(n)$. Consider the time required by a single process. The time for the 2×2 case is $t(2) = 2t_c + t_a + 2t_m$, where t_c is the time for a communication step, t_a is the time for a scalar addition, and t_m is the time for a scalar multiplication. For the $n \times n$ case, the recurrence relation for the time is

$$t(n) = 2t_c + t_a + t_o + 2t\left(\frac{n}{2}\right)$$

where t_o is the overhead time for each recursion level. The closed form is

$$t(n) = (2n-2)t_c + (n-1)t_a + (n-2)t_o + nt_m.$$

Therefore the time for this matrix multiply algorithm using n^2 processes is $O(n)$.

A simple modification and the addition of more processes, produces an algorithm which runs in $O(\log n)$ time. This is achieved by evaluating all $8 \frac{n}{2} \times \frac{n}{2}$ matrix products at the same time, instead of 4 at a time. Figure 3-4 shows the connections needed for an 8×8 matrix product. Each box represents a 8×8 plane of processes. The

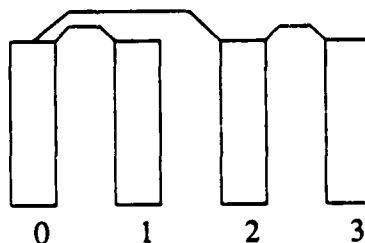


Figure 3-4: 8×8 Connections for $O(\log n)$ Matrix Product

algorithm starts with n^2 processes active, represented by plane 0. These processes contain the original matrices, A and B. The processes communicate as in the $O(n)$ algorithm. At this point, each $\frac{n}{2} \times \frac{n}{2}$ block of processes has two matrix products to compute. One of the products is sent to a set of previously inactive processes, represented by plane 2. The connection shown between plane 0 and plane 2 has a communication channel for each of the processes, connecting corresponding processes in each plane. This doubles the number of active processes. The same algorithm is used to compute the "new" products. After the $\frac{n}{2} \times \frac{n}{2}$ products have been computed, the processes that were sent the second product, send back their result. The results of the two products are added element by element to form the result of the $n \times n$ matrix product. The recursion for this algorithm stops when a 2×2 product is to be computed. Each process does both multiplications and the one addition.

To see that the execution time is $O(\log n)$, consider the time of a single process in plane 0. The 2×2 case has the same time of $t(2) = 2t_c + t_a + 2t_m$. For the $n \times n$ case, the recurrence relation is

$$t(n) = 5t_c + t_a + t_o + t\left(\frac{n}{2}\right)$$

where constants measure the same quantities as before. This recurrence relation is for the original n^2 active processes. The $5t_c$ comes from two t_c 's for the original communication, two t_c 's from sending one subproblem to a "new" process and a t_c for getting the result back from the "new" process. The closed form is

$$t(n) = (5(\log n - 1) + 2)t_c + \log n t_a + (\log n - 1)t_o + 2t_m.$$

This algorithm uses $\frac{n^3}{2}$ processes. It starts with n^2 active processes. After the initial communication, the n^2 processes are divided up into 4, $\frac{n}{2} \times \frac{n}{2}$ sections, each having two matrix products to compute. Every process sends two values, its part of one matrix product, to an inactive process, thus activating it. This doubles the number of processes.

We now have $8, \frac{n}{2} \times \frac{n}{2}$ problems using $2n^2$ processes. Each matrix product is then computed by a "recursive call". This is one recursion level. At each successive recursive level the number of active processes is doubled. There are $\log n - 1$ levels of recursion. This gives $2^{\log n - 1} n^2$ or $\frac{n^3}{2}$ active processes at the evaluation of the 2×2 products.

What we really have is a single algorithm for matrix multiply that uses $\frac{n^3}{2}$ processes and takes $O(\log n)$ time. The $O(n)$ version was just a result of reduced parallelism. Again, this algorithm is not a parallelized version of a standard sequential algorithm.

3.3. The Fast Fourier Transform

There are some sequential divide-and-conquer algorithms that parallelize very easily and produce good results. One of these is the fast Fourier transform (FFT). One use for the FFT is multipoint evaluation of a polynomial over a field F [60]. Given a degree N polynomial,

$$a(x) = \sum_{i=0}^{N-1} a_i x^i,$$

multipoint evaluation computes the value of $a(\alpha_i) = A_i$ for each of m points $\{\alpha_i, 0 \leq i < m\}$.

By choosing $m = N = 2^k$ and $\alpha_i = \omega_i$, where ω is a primitive N th root of unity in F , the FFT algorithm computes all A_i , $0 \leq i < N$ in sequential time of $O(N \log N)$. (Figure 3-5 shows the sequential algorithm [60].) This is based on the decomposition of

$$a(x) = \sum_{i=0}^{N-1} a_i x^i$$

into

```

Algorithm FFT(N,a(x),ω,A)
  if N = 1 then
    A0 := a0;
  else
    /* split */
    n := N / 2;
    b(x) :=  $\sum_{i=0}^{n-1} a_{2i}x^i$ ;
    c(x) :=  $\sum_{i=0}^{n-1} a_{2i+1}x^i$ ;
    /* recursive calls */
    FFT(n,b(x),ω2,B);
    FFT(n,c(x),ω2,C);
    /* combine */
    for k := 0 to n-1 do
      Ak := Bk + ωkCk;
      Ak+n := Bk - ωkCk;
    endfor
  endif

```

Figure 3-5: Sequential FFT Algorithm

$$a(x) = b(y) + xc(y),$$

where

$$N = 2n, y = x^2, b(y) = \sum_{i=0}^{n-1} a_{2i}y^i, \text{ and } c(y) = \sum_{i=0}^{n-1} a_{2i+1}y^i.$$

It also relies on two properties of $\{\omega^i, 0 \leq i < N\}$. The first is that $\{\omega^{2i}, 0 \leq i < n\}$ has exactly n distinct elements, $\{\omega^{2i}, 0 \leq i < n\}$. The second is that $\omega^{j+n} = -\omega^j$. This second property yields $A_j = a(\omega^j) = b(\omega^{2j}) + \omega^j c(\omega^{2j})$ and $A_{j+n} = a(\omega^{j+n}) = b(\omega^{2j}) - \omega^j c(\omega^{2j})$. The problem is then divided up into two instances of multipoint evaluation, for $b(y)$ and $c(y)$ at $\{\omega^{2i}, 0 \leq i < n\}$. Since ω^2 is a n th root of unity, these subproblems can be solved using the FFT algorithm.

To parallelize this algorithm, the split, the sub-problem solution, and the combine are each done in parallel. We are assuming that we have N processes, that a_i is initially in process i , and that the result, A_i , is to be in process i after the algorithm is completed.

The split takes $a(x)$ and produces $b(y)$ and $c(y)$. To allow the sub-problems to be solved in parallel, $b(y)$ and $c(y)$ must be placed in "separate" process groups conforming to the initial conditions. We divide the processes in half and put $b(y)$ in the first half and $c(y)$ in the second half. This split is shown at the top of Figure 3-6. In the figure, each row of processes is the original processes at a different time in the algorithm. The first row is the original configuration and the last row is the final result. The lines

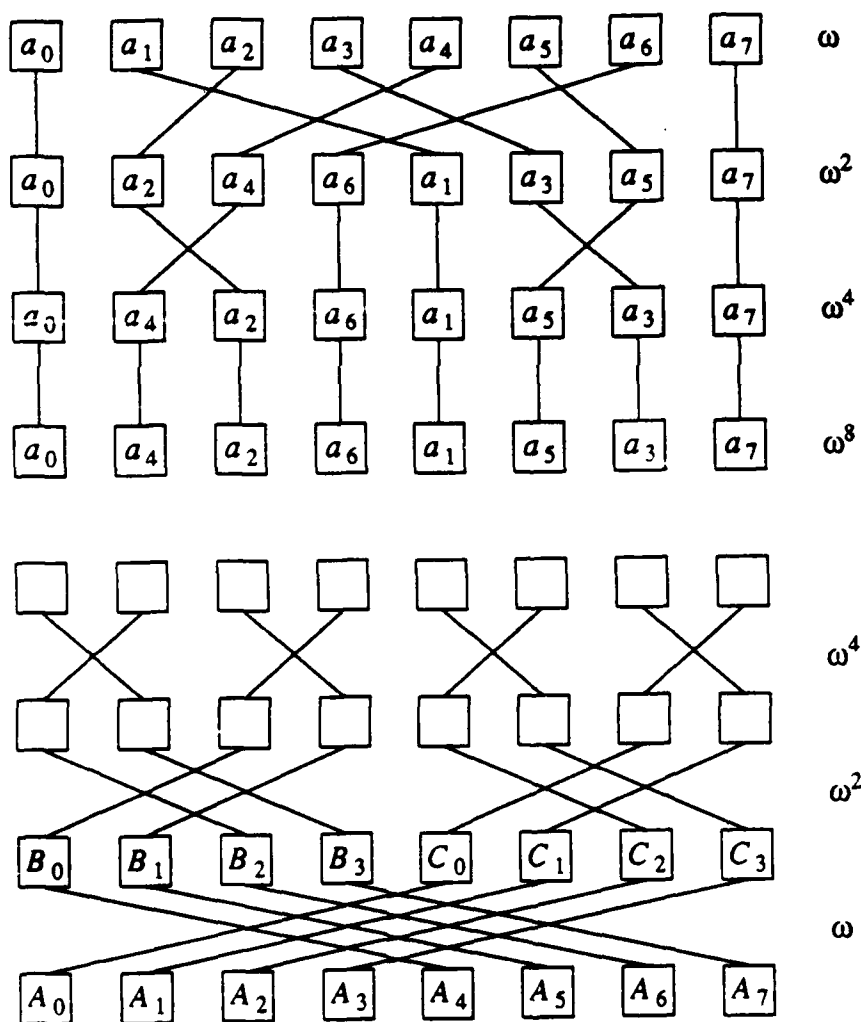


Figure 3-6: Parallel FFT data movement

between rows shows data movement between time frames. To accomplish the split, the a_i values are unshuffled in one communication step. After this data movement, the split is complete and each half of the processes has an independent problem and therefore can be solved in parallel.

After both subproblems are solved, the first half of the processes contain the B_i s and the last half contain the C_i s. Corresponding processes in the halves exchange their values as shown at the bottom of Figure 3-6, retaining a copy of their own value. Process i in the first half computes $A_i \approx B_i + \omega^i C_i$ and process i in the last half computes $A_{n+i} = B_i - \omega^i C_i$.

Figure 3-7 shows the algorithm for a single process involved in the parallel FFT algorithm. The parameter N is the current problem size and i is the processes' position in that group of N processes. Figure 3-6 shows the complete algorithm in terms of communications for a FFT of size 8. Notice that the bottom half shows the n-cube connections. The upper half is the unshuffle connections on blocks with 2^k processes, for $1 < k \leq \log n$. This parallel version of the FFT takes $O(\log n)$ time. This comes from the fact that one communication was needed for both the split and combine steps and that the recursion goes $\log n$ levels deep.

3.4. Divide-and-conquer and the n-cube

As we noticed earlier, sequential divide-and-conquer often is related to the tree structure. This tree structure is the method in which the data is referenced. Since data reference in sequential paradigms is related to communication structures in parallel paradigms, we might expect a single interconnection structure related to parallel divide-and-conquer. At first glance we might assume that this structure will also be a tree, which has a communication bottleneck at the root. But, as we have seen in the divide-and-conquer algorithms studied so far, the n-cube interconnection structure has been part of or the complete interconnection structure.

The obvious question at this point is "What is the common feature of these algorithms that produce the n-cube interconnection structure?" Is it the divide-and-conquer

```

Algorithm FFT(N,i,a,ω,A)
  if N = 1 then
    A := a;
  else
    /* split */
    PEunshuffle(N) ← a;
    a ← PEunshuffle(N);
    /* sub problem */
    n := N / 2;
    FFT(n,i mod n,a,ω2,B)
    /* combine */
    if i < n then
      PE*,+n ← B;
      C ← PE*,+n;
      A := B + ωiC;
    else
      C := B;
      PE*,-n ← C;
      B ← PE*,-n;
      A := B - ωi-nC;
    endif
  endif
endif

```

Figure 3-7: Single Process, Parallel FFT Algorithm

paradigm or a combination of the paradigm and the specific algorithms? As we have seen, the interconnections for divide-and-conquer algorithms come from the divide and combine steps. The complete interconnection is defined by the all divide and combine steps in the entire algorithm. In looking at the three previous divide-and-conquer algorithms, we notice that the divide step in the matrix multiply and the second occurrence of divide-and-conquer in Batcher's sort and that the combine step in the FFT require communication between corresponding processes in each of the sub-problems. It is easy to see that the main problem's divide or combine requires the high order edges of the n-cube. Because of the recursive nature of divide-and-conquer, each sub-problem will require a similar sub-structure, yielding the binary n-cube.

The other feature that is needed to yield the binary n-cube is that the original problem starts with all processes "active." In each of the previous algorithms, the problem is

distributed in n or n^2 processes. All n or n^2 processes communicate in the divide or the combine step. As an example of an algorithm that does not start with all processes active, consider the numerical integration algorithm discussed in Chapter 2. We change that algorithm to a divide-and-conquer algorithm.

A single process has the interval that needs to be integrated. The process divides the interval into two intervals and sends one of the two sub-problems to a previously inactive process. Each active process now integrates its interval using the same algorithm. The recursion is stopped after the first process has recursed $\log n$ times, yielding n active processes. After the recursion has stopped, each process integrates its interval using standard methods. The combine steps are then adding the areas of the two sub-intervals or recording that they are not accurately integrated and added to the list of sub-intervals needing more processing.

Figure 3-8 shows the communication links for $n = 16$. The initially active process is numbered 0. Processes activated by the i th level of recursion are numbered i . Notice that this is not the binary tree one might expect, but a "tree" constructed from edges of the binary n -cube. Processes labelled i have all binary n -cube connections of order $\log n - i$ and less. Notice that this is also the same interconnection structure found in the "3rd" dimension of the $O(\log n)$ matrix multiply.

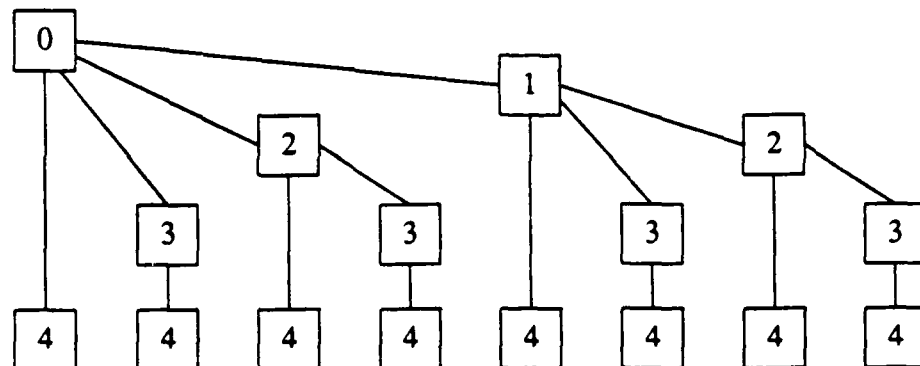


Figure 3-8: DAC Integrate Connections

3.5. Connected Ones

Not all divide-and-conquer algorithms require the binary n -cube interconnection structure. Stout discusses several divide-and-conquer algorithms for image processing [85]. One of these is the algorithm for connected components for image data, sometimes called connected ones [62, 72]. The divide-and-conquer strategy used to solve this problem produces the 4-neighbor mesh.

Given an $n \times n$ black and white pixel image, each connected set of black (or white) pixels are to be assigned a unique number. Two pixels are connected if they are neighbors using either the 4-neighbor or 8-neighbor methods. The divide-and-conquer solution takes the image and divides it into 4 quadrants, solving the connected ones problem for each quadrant independently. Once they are solved, only the boundaries with other quadrants need to be examined to find components that cross the boundary. The results of this border examination is a graph connected components problem where the vertices of the graph are components in each of the four quadrants and the edges are defined by having the components adjacent on the boundary. The solution to the graph connected components provides the final labelling.

For a parallel implementation, we use a $n \times n$ process array where each process contains one pixel. The division into quadrants provides four independent problems to be solved in each $\frac{n}{2} \times \frac{n}{2}$ sub-arrays of the processes. After the four sub-problems have been solved, the boundaries need to be examined to find components that lie in two or more quadrants. Adjacent processes on the boundaries exchange information as shown in Figure 3-9. It shows the connections required for the final boundary exchange for an 8×8 image. These boundary processes now know which components in each quadrant are adjacent. This information is a graph with $O(n)$ edges and $O(n)$ nodes. The $n \times n$ processes then solve the graph connected components problem. Using the solution to the graph connected components, each quadrant corrects its label. The complete graph for all boundary exchanges is the 4-neighbor mesh. The algorithm used for solving the graph connected components adds other edges.

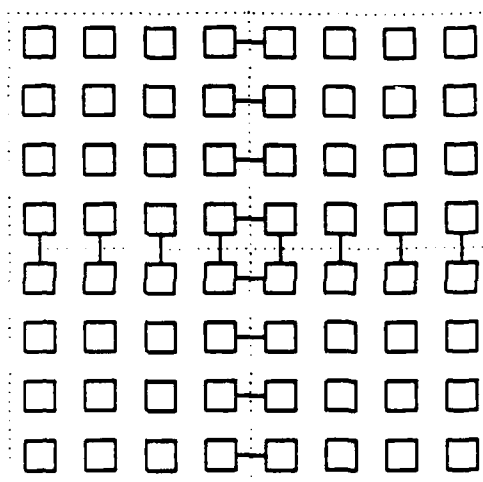


Figure 3-9: Connected Ones Communication

Nassimi and Sahni [62] provide a complete algorithm using only the 4-neighbor mesh that runs in $O(n)$ time. Prasana Kumar and Eshaghian [72] give a sketch of the algorithm for a mesh of trees that runs in $O(\log^4 n)$ time.

3.6. Contraction of Divide-and-Conquer Algorithms

We would also like to apply contraction to the divide-and-conquer paradigm. Since we have been applying contraction to specific interconnection structures, a natural interconnection structure to study for the divide-and-conquer paradigm is the n -cube.

The first algorithm we will consider for contraction is the $O(n)$ matrix multiply algorithm. To be able to talk about the contraction we need to be able to talk about the n -cube. An *order k* n -cube has 2^k processes. (See Figure 3-10.) We assume that the processes are numbered in a row major order. Each process is an order 0 cube, each pair of processes, $2i$ and $2i+1$ is an order 1 cube, and similarly for each block of 2^j processes for $1 \leq j < k$. An edge is an *order k* edge if it connects corresponding processes in order $k-1$ cubes and therefore connects processes whose numbers differ by 2^{k-1} .

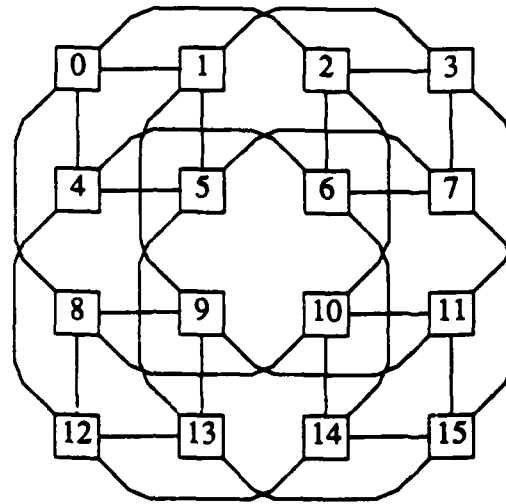


Figure 3-10: An Order 4 N-cube

The $O(n)$ time matrix multiply algorithm runs in an order $2 \log n$ cube. Since the processes are numbered in row major order, the matrices are stored in the processes in row major order. The longest edge connecting processes containing the same row of data is an order $\log n$ edge. To identify this edge easily, we call it the order k edge. That makes the longest edge connecting processes in the same column an order $2k$ edge, the longest edge in the order $2 \log n$ or order $2k$ cube.

To find the cost of the matrix multiply algorithm, $K(CMM)$, we need to find the edge with the most messages. At the first level of recursion, the order k and $2k$ edges were used to send a message each way. This is the only use of these edges in the algorithm. Therefore, $w(e) = 1$, where e is a order k or $2k$ edge. At the second level of recursion, two matrix products are computed using the order $k-1$ and $2k-1$ edges. Each matrix product sends one message each way on each edge giving $w(e) = 2$, where e is a order $k-1$ or $2k-1$ edge. At level l of the recursion, $w(e) = 2^{l-1}$ messages over the order $k-(l-1)$ and $2k-(l-1)$ edges. The recursion stops when we have order 2 cubes. This is at the $\log n$ level of recursion. There are $\frac{n}{2}$ matrix multiplies done by order 2 cubes. These order 2 cubes use the order 1 and $k+1$ edges. Each matrix multiply sends

1 message each way giving $w(e) = \frac{n}{2}$, where e is a order 1 or $k+1$ edge. Since this is the largest value, $K(CMM) = \frac{n}{2}$.

Consider any contraction, $M(CMM, p)$ where $p = 2^m$ for some $m \leq 2k$. $M(CMM, p)$ will map $\frac{n^2}{p}$ logical processes to every processor. This allows us to put a cube of order $\log \left[\frac{n^2}{p} \right] = 2k - m$ into each processor. The processor-to-processor connection graph is also a cube and is of order m . Each processor-to-processor connection supports $\frac{n^2}{p}$ communication paths in the original graph. The real question is which sub-cube do we map to each processor. The cost of the contraction, $K(M(CMM, p))$ will be $\frac{n^2}{p}$ times the maximum $w(e)$, where e is mapped to a physical edge. If e is order 1 or $2k+1$ from the original cube, $K(M(CMM, p)) = \frac{n^3}{2p}$.

Consider the contraction that maps the edges of order 1 through $\left\lfloor \frac{2k-m}{2} \right\rfloor$ and order $k+1$ through $k + \left\lfloor \frac{2k-m}{2} \right\rfloor$ into internal edges. (This coalesces square blocks of processors when the cube is laid out in a row major order as in Figure 3-10.) This makes the edge of order $k + \left\lfloor \frac{2k-m}{2} \right\rfloor + 1$ the edge with the most messages. This edge is used by level $k - \left\lfloor \frac{2k-m}{2} \right\rfloor$ of the recursion. From before we know that $w(e) = 2^{k - \left\lfloor \frac{2k-m}{2} \right\rfloor - 1} = \frac{\sqrt{p}}{2}$. Therefore $K(M(CMM, p)) = \frac{n^2 \sqrt{p}}{2p}$. Clearly, this contraction is better in terms of the number of messages over the busiest physical edge than any contraction that does not keep the high traffic logical edges internal to a processor.

By contrast, let us consider the Batcher bitonic merge sort. This sort runs on an order k cube to sort $n = 2^k$ elements. The final sorting will have the smallest element in the first process and the largest element in the last process. Figure 3-11 shows a graphical representation of the algorithm. (This is the same as Figure 3-1.) The arrows represent a data exchange and a compare, leaving the larger number at the end with the arrow and the smaller at the other end. It is obvious from the figure that the order 1 edge has the most messages. Therefore, $K(\text{SORT}) = \log n$.

Again, to contract this algorithm, we see that we want to assign a sub-cube into a processor. Consider the contraction $M(\text{SORT}, p)$ where the edges of order 1 through order $\log p$ are mapped to internal edges. We are assuming that $p = 2^m$, for some $m \leq \log n$. This contraction assigns the busiest logical edges to be internal edges. These edges carry $\log n - \log p$ messages. Since each processor contains $\frac{n}{p}$ logical processes, $K(M(\text{SORT}, p)) = \frac{n(\log n - \log p)}{p}$. Any contraction that does not map these first $\log p$ edges to internal edges will have a higher communication cost. These results agree with and explain the results of Hsiao[47], even though his final algorithm

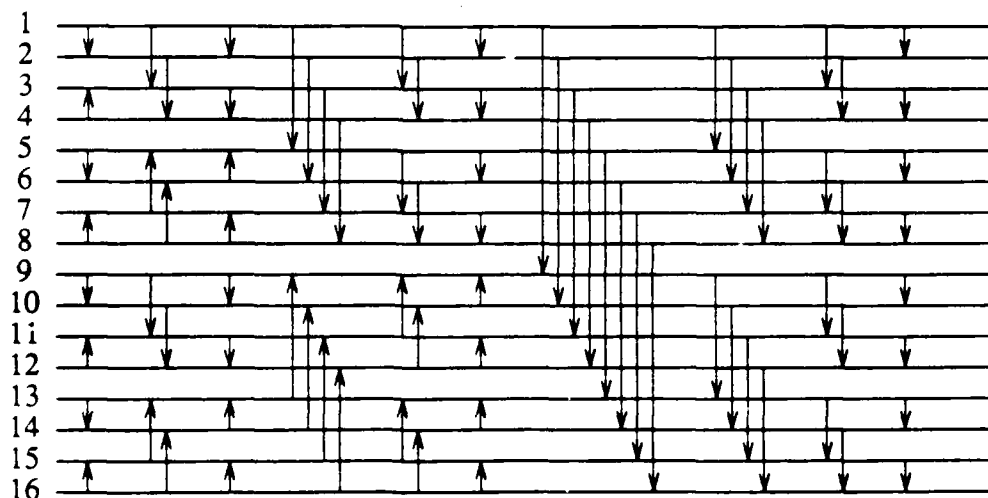


Figure 3-11: Batcher's Bitonic Merge Sort

was embedded in a grid instead of another cube.

In comparing the contractions for matrix multiply and Batchers's sort, we see that the same size cube is mapped in a different way when mapped to the same number of processors. The busiest edges are different for the two algorithms, thus, the contractions are different.

CHAPTER 4

The Pipelining and Systolic Paradigms

Perhaps the most widely used paradigm for nonshared memory parallel computation is the systolic approach [55]. Although definitions of systolic computing have been provided by the inventors [53, 58] and others [59], the concept remains imprecise. We present the pipelining paradigm and show that the systolic paradigm is a special case of the more general concept of pipelining. We examine the paradigms using examples of both systolic algorithms and pipelined algorithms that do not seem to meet all of the criteria of the systolic definition. We conclude by considering the contraction of pipelined algorithms.

4.1. Pipeline and Systolic Definitions

Pipelining is a well known technique used in hardware design [8, 16]. These pipelines, by definition, exhibit parallel processing. Most classical pipelines are linear, having one stream of data that visits all processors in succession. We broaden the pipelining definition to include multiple data streams that may interact. We also want to look at the algorithmic structure of these pipelines, not their hardware implementations.

The key paradigmatic concept in pipelining and systolic algorithms is the decomposition of the problem into subcomputations that are assigned to dedicated processes with the data "flowing" through the processes, visiting all or an appropriate subset of processes to complete the computation for that input. In addition, systolic algorithms are expected to exhibit additional properties: locality of communication, a regular communication structure, and have only a few different types of simple processes. These extra properties come from the desire to implement the systolic algorithm directly in custom VLSI. The locality of communication requires that two processors which are directly connected must be $O(1)$ distance apart. The other properties are desirable for

the ease of implementation.

We can again consider the execution time for members of the paradigm regardless of the algorithm. The cost, l , called *latency*, is the longest time required for a single piece of data to flow through the processes. This is the minimum time for completion of a pipeline algorithm. The number of input groups, t , gives the amount of data that needs to be processed. Generally, these two measures determine the running time of $O(l+t)$.

4.2. Flow Test

The property of flow not only gives these types of algorithms their names, but it seems to be the property that distinguishes them from other parallel algorithms with similar properties. Specifically, in Chapter 2 we described numerical CAB algorithms exhibiting many of the properties of the pipelined algorithms. The Jacobi iterative algorithm was decomposed into many small (approximation) subcomputations, each process was specialized (to computing the voltage at a point), the communication was regularly structured and local, and there were a small number of processes (boundary, source and interior). But such algorithms are not normally designated as pipelined or systolic. Assuming that this observation is a reflection of the style of computing and not simply a statement about the historical order of the development of the Jacobi iteration and systolic computation, then what seems to be lacking in the Jacobi is the concept of flow.

The claim that the communication of the Jacobi algorithm does not exhibit flow and that, say, the Kung and Leiserson systolic matrix multiplication algorithm does is obviously a statement to which not all researchers would subscribe, but consider the following "flow test" which systolic algorithms generally pass and the Jacobi fails:

Flow Test [66]: Select an arbitrary communication edge and "radioactively tag" a single transmission across that edge. (The concept of radioactive tags is due to Cunny and her students, where it is used in debugging parallel programs [21].) As the computation progresses from the time of tagging, let all values computed with one or more radioactive values become radioactive. Then define the "contaminated region" as the set of processes and communication channels touched

by some radioactive value. An algorithm passes the flow test if the edge over which the initial value was transmitted is on the boundary of the region; otherwise it fails the test.

Clearly, the Flow Test captures the concept of flow in that the tagged value "flows out" to define the contaminated region. The Jacobi algorithm fails the Flow Test because a process that receives the radioactive value will be sending out a radioactive element to its four neighbors, eventually contaminating the entire process array, regardless of which was the originally tagged edge. When it is presented, we will see that the Kung and Leiserson band matrix multiply algorithm passes the Flow Test. The Flow Test is not a perfect test, since some algorithms generally called systolic will fail, *e.g.* certain linear arrays such as the band matrix-vector multiplication and the band lower triangular linear system solver of Kung and Leiserson [52], but it does seem to identify the presence of flow.

4.3. Systolic Algorithms

We start first with the systolic algorithms. They are good examples of several aspects of pipeline algorithms besides being systolic. For each algorithm, we show the interconnection structure, describe the algorithms, and apply the Flow Test to the algorithm.

4.3.1. Band Matrix Multiply

One of the first systolic algorithms developed was the band matrix multiplication of Kung and Leiserson [52] shown in Figure 4-1. The algorithm multiplies two $n \times n$ matrices, A and B , with band widths l and m respectively, producing C . The processes are organized as an $l \times m$ hex-connected array. These matrices flow in different directions through the processes. To produce the correct result, the placement of the input data is very important. Since all data items move at the same time, there must be two "empty" spaces, or "bubbles," between the items. This insures that the correct data items meet at a process, *e.g.* a_{11} , b_{11} , and c_{11} in the figure. The C values are initialized to zero as they "enter" the array and therefore no input is required. Each data path can be

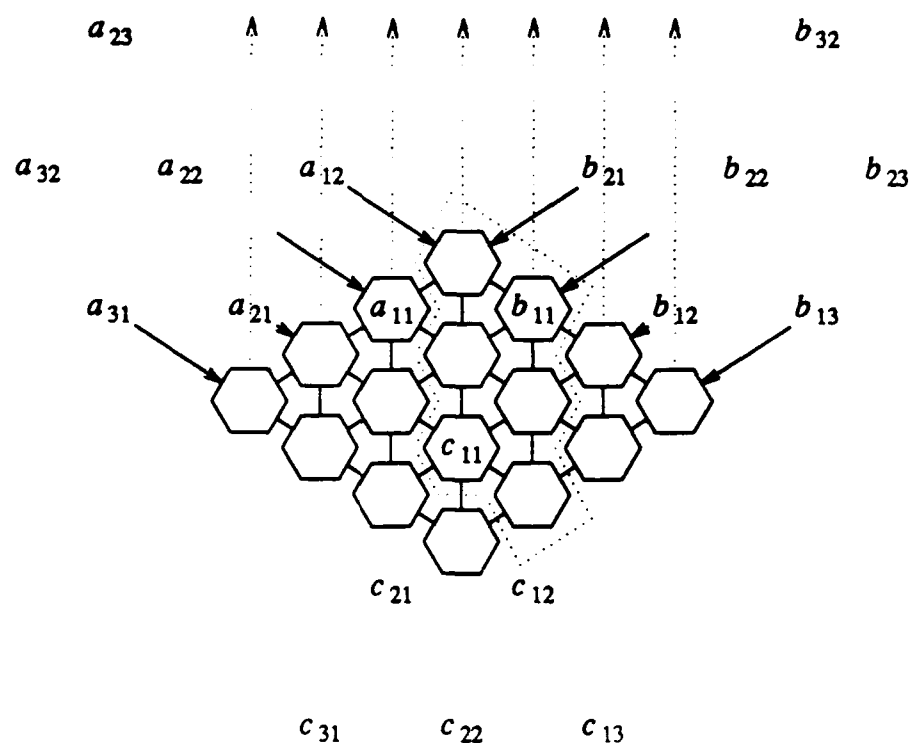


Figure 4-1: Band Matrix Multiply

considered as a pipeline that interacts with the other pipelines. Notice that the results do not exit from the end of the input pipelines, but from output pipelines that cross and interact with the input pipelines. The time required is $O(n)$.

This algorithm passes the Flow Test. Let us tag a_{21} as it crosses the edge coming into the process containing c_{11} in Figure 4-1. The values a_{21} , b_{12} and c_{22} will meet at the process containing c_{11} . c_{22} becomes radioactive because it is now a result of an expression containing a radioactive piece of data. Notice that b_{12} does not become radioactive, even though it flows through a contaminated process. c_{22} contaminates all processes above as it flows up. In a similar fashion, c_{23} becomes radioactive and contaminates its column of processes. The final contaminated area is the two output pipelines above the travel of a_{21} and the original edge is on the boundary, and thus the algorithm passes the Flow Test.

4.3.2. WAP Matrix Multiply

Some algorithms do not produce results that flow out of the pipelines. Consider the matrix multiply algorithm of S. Y. Kung *et al.* [54] for the wavefront array processor. This algorithm uses n^2 processes to multiply two $n \times n$ matrices. Figure 4-2 shows the data staging and the interconnection structure. The data arrives at every "clock cycle" and flows in vertical and horizontal pipelines. All c values are initialized to zero. As the data flows by each process, the a value times the b value is added into the c value. When the input data has passed through the input pipelines, the correct c values are contained in the processes. Although the results do not flow out of the pipelines, we nevertheless consider this to be a systolic algorithm. Because there are no results flowing in this algorithm and the inputs flow in a single row or column, it trivially passes the Flow Test.

The advantage to this algorithm is that no "bubbles" are required in the input pipelines. It does, however, require the data to be staged correctly so that the proper a and b

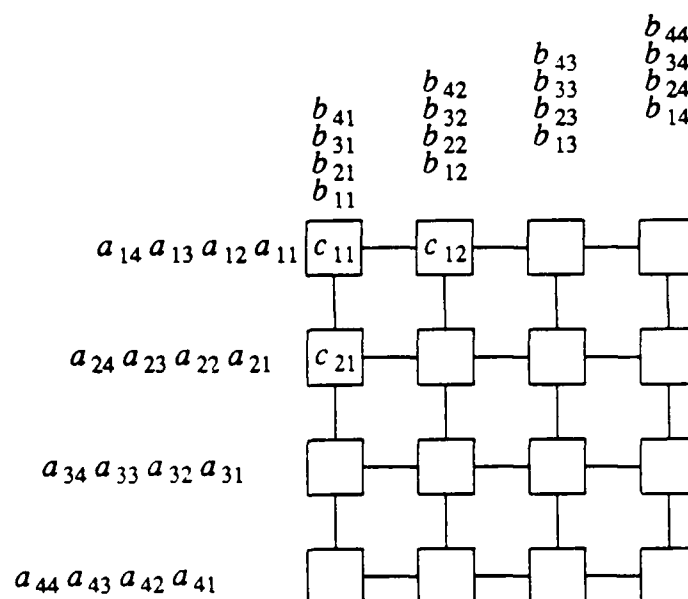


Figure 4-2: Data Staging for the WAP Matrix Multiply

values meet at the correct process. Also, if results are needed externally, the final values must be shifted out of the process array. The algorithm requires $O(n)$ time.

4.3.3. Dynamic Programming

Another interesting systolic algorithm is the dynamic programming algorithm of Guibas, Kung and Thompson [38]. The processes are arranged in a triangle with pipelines in the horizontal and vertical directions. (See Figure 4-3.) Data moves from the diagonal toward the top and right. At time $2t$, results are ready at every process that is t distance away from the diagonal. Processes (12), (23), etc. are defined to be distance 1 from the diagonal. For the next t time units these results move up and to the right at the rate of one process per time unit. After that, they move one process in the same direction every two time units. Depending on the problem to be solved, the processes may have preloaded data. The computation performed at each process will also vary depending on the problem to be solved.

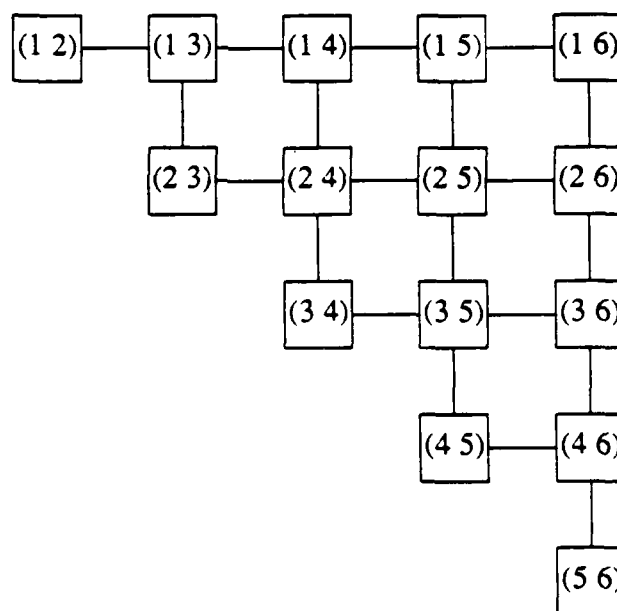


Figure 4-3: Dynamic Programming Process Structure

An interesting part of the Guibas, Kung and Thompson algorithm is the use of the pipelines for both fast moving data elements and slow moving data elements. This feature provides an order reversal on the data, and it can be implemented either by a pair of connections between the processes or by a single connection doing double duty. This algorithm also passes the Flow Test because all data flows up or to the right.

4.3.4. Lower Triangular Linear System Solver

Our last systolic algorithm is the lower triangular linear system solver of Kung and Leiserson [52]. Given the linear system $Ax = b$, where $A = (a_{ij})$ is a nonsingular $n \times n$ band lower triangular matrix and $b = (b_1, \dots, b_n)^T$, we want to solve for $x = (x_1, \dots, x_n)^T$. The following recurrences can be used to solve for x :

$$y_i^{(1)} = 0 \quad ,$$

$$y_i^{(k+1)} = y_i^{(k)} + a_{ik}x_k,$$

$$x_i = (b_i - y_i^{(i)})/a_{ii}.$$

Figure 4-4 shows the linear systolic algorithm implementing the recurrences for a band width of four. The box processes are implementing the first two equations and the round process is implementing the third equation. Notice that the x_i and y_i values do not need to come from an external source. Notice that this algorithm also has "bubbles" in the input and output data.

This algorithm does not pass the Flow Test. Tag any edge between processes and shortly all processes become contaminated. This is because the outputs at each processor are results of computations of both inputs. Regardless of which input is radioactive, both outputs are radioactive. The initial edge is no longer on the boundary of the contaminated area. But this is a systolic algorithm. It is very obvious that data flows in several directions in this algorithm. The Flow Test will fail for any pipelined algorithm with two directional pipelines that has results flowing in both directions.

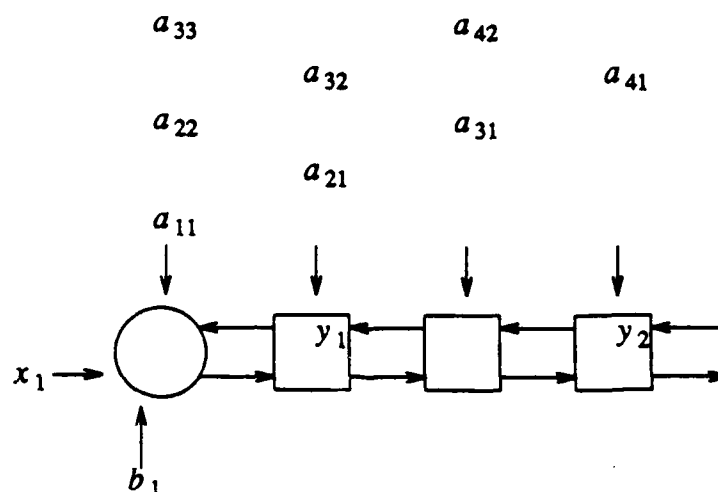


Figure 4-4: Systolic Linear System Solver

4.4. Other Algorithms

We now look at a few algorithms that are pipelined algorithms but may not satisfy all of the systolic definitions. For each algorithm, we give the reason they are not systolic algorithms.

4.4.1. Vector Sum Algorithm

Consider the problem of computing the sums of n element vectors. (See Figure 4-5.) A single vector is summed using a tree of $\frac{n}{2}$ processes. The vector is input at the leaves and the sum of the vector comes out from the root. Successive vectors are input at the leaves when the previous vector has moved up the tree by one level. We can identify each level in the tree as a stage in the pipeline, with the data flowing from stage to stage. The latency is $O(\log n)$, yielding an execution time of $O(d + \log n)$ where d is the number of n element vectors to be summed. This algorithm passes the flow test and it appears to have the features of systolic algorithms. But trees cannot be laid out in the plane with edges of constant length independent of the size of the tree [69]. Thus, if communication time is proportional to distance, as is the case in the VLSI models normally associated with systolic arrays, then the time between stages increases as larger

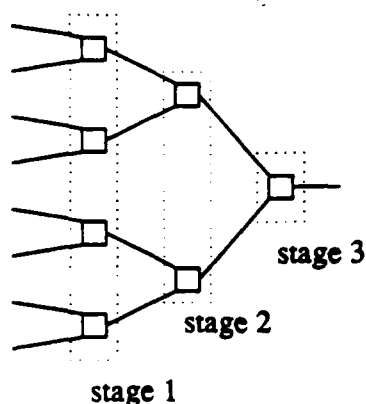
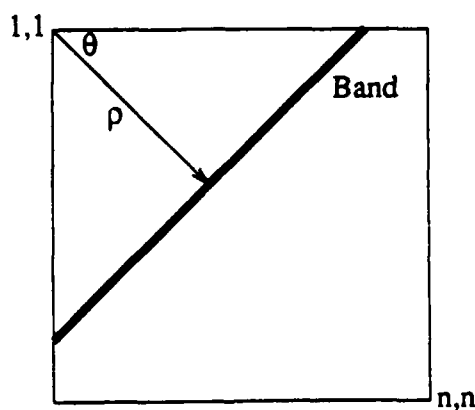


Figure 4-5: Vector Sum Pipeline

and larger trees are considered. One wonders, therefore, whether tree based pipeline algorithms meet the locality requirement of systolic computation.

4.4.2. Hough Transform

Another pipeline algorithm is the Hough Transform algorithm designed by Cypher and Sanz [22]. The Hough Transform can be used to detect lines and curves in picture data [26]. Given a $n \times n$ image of pixels, the Hough transform does computations on bands, usually one pixel wide. (See Figure 4-6.) For a given θ , the complete picture will

Figure 4-6: Hough Transform on $n \times n$ Picture

be processed by computing all bands. The bands are identified by the pair (θ, ρ) . Most applications using the Hough transform require results for multiple values of θ .

The algorithm uses n^2 processes connected by the $n \times n$ torus. (A $n \times n$ mesh with end around connections.) The picture is distributed one pixel per process. A value of θ is input to the first column. (Actual implementations may use $\sin \theta$ and $\cos \theta$ as input.) The input, along with the values being computed will move to the right across the processes. All bands for one θ will be calculated by a single sweep. After the first value of θ has moved to the second column, another value of θ may be input.

Consider a single value of θ . Each process will contribute its pixel value to one band. The band number is calculated by $[row * \cos \theta + column * \sin \theta]$. For $\theta = 90^\circ$, each row of processes will contain a single band. For other values of θ , the bands will not be horizontal. Figure 4-7 shows the band numbers for $\theta = 67.5^\circ$. To keep the bands more horizontal than vertical, θ is limited to $45^\circ \leq \theta \leq 135^\circ$. For the other values of θ , the algorithm can be rotated by 90 degrees. For $\theta < 90^\circ$ the bands move up in the successive columns. To simplify our discussion we assume that $45^\circ \leq \theta \leq 90^\circ$.

There is a single total for each band. This total is started in column 1 and the band number is then associated with the total. In the case where two processes in the same column contain data from one band (rows 5 and 6 in Figure 4-7), the lower processor retains the band total and does all the computation. To allow for this case, the processes initially shift all pixels down by one process. As a total for a band moves through the processes, the accumulated band data follows the bands by moving up in the column. The limit of $45^\circ \leq \theta$ guarantees that each band moves up by at most one process.

If a band total moves off the top of the processes the band is completed, but the data needs to be moved to the last column. This is accomplished by making the data passive as it arrives at the last row. The passive data continues to shift up in the column to make sure that any one process has the maximum of one passive data element. Also, as the band total moves up out of the last row, new band totals must be started. This means that each process contains data for two bands, the active band involved in the

1,1

1	1	2	2	2	3	3	3	4	4	5	5	5	6	6	7
2	2	2	3	3	4	4	4	5	5	6	6	6	7	7	7
3	3	3	4	4	5	5	5	6	6	6	7	7	8	8	8
4	4	4	5	5	5	6	6	7	7	7	8	8	9	9	9
5	5	5	6	6	6	7	7	8	8	8	9	9	9	10	10
5	6	6	7	7	7	8	8	8	9	9	10	10	10	11	11
6	7	7	7	8	8	9	9	9	10	10	11	11	11	12	12
7	8	8	8	9	9	10	10	10	11	11	11	12	12	13	13
8	9	9	9	10	10	10	11	11	12	12	12	13	13	14	14
9	10	10	10	11	11	11	12	12	13	13	13	14	14	14	15
10	10	11	11	12	12	12	13	13	13	14	14	15	15	15	16
11	11	12	12	12	13	13	14	14	14	15	15	16	16	16	17
12	12	13	13	13	14	14	15	15	15	16	16	16	17	17	18
13	13	14	14	14	15	15	15	16	16	17	17	17	18	18	19
14	14	15	15	15	16	16	16	17	17	18	18	18	19	19	19
15	15	15	16	16	17	17	17	18	18	18	19	19	20	20	20

16,16

Figure 4-7: Band Numbers for $\theta = 67.5^\circ$

local computation and the passive band that does not pass through the current process.

Notice that in a single column, not all band totals will move at the same time. Consider column 4 in Figure 4-7. The totals for bands 2 through 6 and 15 move up by one process, the totals for bands 7 through 14 remain in the same row, and a new band, band 16, is started in the last row. This movement causes the process in row 6 to have no active data. These "holes" are needed where one row moves up and another one does not as is the case with bands 15 and 14 respectively.

The time used by this algorithm depends on two variables, n , the picture size and p , the number of values of θ . One value of θ is calculated by a single sweep of the data across the $n \times n$ processes and requires $O(n)$ time. Since successive values of θ can be

input at regular intervals, the last value of θ will be input at time $O(p)$. Therefore, the total time is $O(n+p)$.

There are two features of special interest. First, the vertical movement of the data is very input data dependent, unlike the other algorithms where the flow pattern is fixed. Each value of θ will exhibit a different pattern of up shifts. Even with this vertical movement, the algorithm passes the flow test. Also, computation involves only one of two data values being moved, swapping the active and passive data when moved off the top or bottom. The final results may have one or two computed values, and this is also input data dependent. Even with these features, it is easy to see that this is a pipelined algorithm. It may be too irregular in other respects to be classified as a systolic algorithm.

4.4.3. Funneled Pipelines

Our last example of pipelined algorithms are the funneled pipelines [43]. The funneled pipelines are used to implement a minimum spanning forest algorithm, a biconnected components algorithm, and several other graph theoretic algorithms. These are interesting because each stage in the pipeline is a *filter*, yielding a pipeline of filters. Each filter outputs half as much data as was input. This produces less data for each stage of the pipeline, allowing later filters more time to do their processing.

The funneled pipeline for the minimum spanning forest is composed of filters that essentially compute reduced minimum spanning forest problems. Each filter is composed of a tree of processes with n leaf processes. These leaf processes are the ones that input and output the data. Input to the pipeline is an upper triangular matrix of edge weights. The filters input two rows of edges and output one row of edges. This is a combination of the two input rows. The tree structure of the filter allows "communication" between the leaves. The root processes contain memory to hold $O(n)$ edges as part of the final result.

The pipeline contains $\log n$ filters, connected by corresponding leaf nodes in each filter. The n rows of the adjacency matrix are input to the first filter. The first filter

passes $\frac{n}{2}$ rows to the next filter, and so forth. After all stages have stopped, the root processes contain $O(n \log n)$ edges that make up a small superset of the minimum spanning forest. A sequential algorithm computes the final result.

There are two specific points of interest for pipelined algorithms. First, this is the only pipelined algorithm that uses communication internal to a pipelined stage to produce the correct results. This complex stage, using internal communication is the non-systolic structure in the funneled pipelines. The second point is the exponentially decreasing data between stages of the pipeline. We saw a similar feature in the vector sum pipeline. The difference is that the vector sum has an exponential decrease in edges, yielding the exponential decrease in data but still retaining the same number of data elements per edge, but the funneled pipelines are connected with the same number of edges and the actual amount of data across these edges reduces exponentially. This is also a feature showing non-systolic properties because systolic algorithms generally have the same amount of data across each edge. The result of this decrease in data is an exponential time delay between the first stage and the last stage, giving a latency of 2^k for a k stage pipeline. This yields a latency of $O(n)$. This does not slow down the algorithm because the first stage requires $O(n)$ time to input the adjacency matrix. The total time is then $O(n)$ time to produce the reduced minimum spanning forest problem.

4.5. Contraction of Pipeline Algorithms

A recurring interconnection structure in systolic algorithms is the mesh, seen in several variations. For contraction of pipelined algorithms we consider the contraction of mesh algorithms. We use the matrix product algorithm for the Wavefront Array Processor (WAP) [54] as our example. Recall that it uses n^2 processes connected in the 4 neighbor mesh for the $n \times n$ matrix product $AB = C$. The data is fed in along the top n processes and from the left n processes, accumulating the final result in each process. A loop is executed n times that reads an A value from the left and a B value from above, multiplies them together, and adds the result to c_{ij} . The A and B values are sent to the right and down, respectively. This causes the upper left process to be the first process to

start execution. As the data moves into the array, there is a wavefront of executing processes on the cross diagonal. Each edge is used to send all of one row of A or one column of B . For the WAP algorithm we have the cost $K(WAP) = n$.

Consider the contraction in Figure 4-8. Let us call this contraction $M_1(WAP, p)$. This is the contraction done by cutting the graph into p equal size connected subgraphs and assigning one process from each subgraph selected from corresponding positions to a single processor. The physical connection graph, shown in Figure 4-8, is a grid with end around (i.e. toroidal) connections. The curved lines use the end around connections. For each logical process in a physical processor, there are horizontal and vertical communication paths. Since we have $\frac{n^2}{p}$ logical processes in a processor, the number of logical edges using one processor-to-processor connection is $\frac{n^2}{p}$. Since all horizontal and vertical edges have the same number of messages, n , we have $K(M_1(WAP, p)) = \frac{n^3}{p}$.

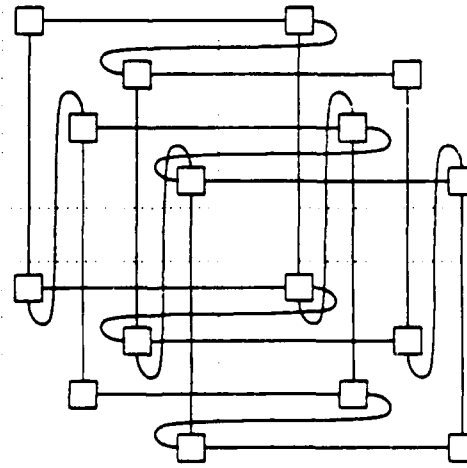


Figure 4-8: A Contraction of 16 Processes to 4 Processors

Consider the contraction in Figure 4-9. Let us call this contraction $M_2(WAP, p)$. This is the contraction done by cutting the graph into p equal size connected subgraphs and assigning an entire subgraph to a processor. We see that only the perimeter processes have edges that go from processor-to-processor. Also, notice that no end around connections are needed. The number of communication paths over one processor-to-processor connection is $\sqrt{\frac{n^2}{p}}$. Each communication path requires n messages giving $K(M_2(WAP, p)) = \frac{n^2}{\sqrt{p}}$.

Comparing the two contractions, we see that $K(M_2(WAP, p))$ is smaller than $K(M_1(WAP, p))$ by a factor of $\frac{n}{\sqrt{p}}$, telling us that M_2 is expected to be the better contraction. We conjecture that M_2 is the best contraction that can be achieved for grid algorithms. The basis for this conjecture is that this contraction has the smallest perimeter for a given area, and has been commonly used for contraction in published algorithms, for example for the Jacobi iterative method [1] and for the conjugate gradient method [34].

Both M_1 and M_2 were programmed using Poker. Table 4-1 summarizes the results of the timings. As predicted, M_2 was the faster contraction, but because the

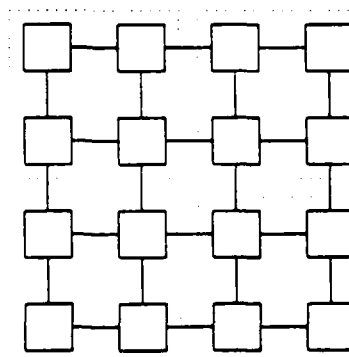


Figure 4-9: Another Contraction of 16 Processes to 4 Processors

Table 4-1: Contraction Timings for the WAP Matrix Multiply Algorithm

WAP matrix multiply: ticks for n (items) on p (processors)				
Contraction	16 on 4	64 on 16	64 on 4	256 on 16
M_1	48854	111478	400542	901543
M_2	31113	73088	221545	707646

communication time is not the only time consuming part in these algorithms the difference is perhaps not as dramatic as might be expected. Again, Appendix A contains the code for both contractions.

CHAPTER 5

Other Paradigms

In this chapter we look briefly at two other techniques. We present these in the belief that they are important and deserve more attention. The first of these is the reduction paradigm, a well known technique used in the theory of computation [44]. We show how this technique applies to parallel algorithms and give several example algorithms using reduction. The second is a way of approaching arbitrary communication algorithms. These algorithms, which may require communication between any two processes, provide problems for CTA machines.

5.1. Reduction

Reduction, sometimes called reducibility, is used in complexity theory for solving decision problems. Simply stated, "If there is an algorithm for deciding X , then there is an algorithm for deciding Y [56]." Often, there is no known algorithm for X , yielding a result that says that deciding Y is no "harder" than deciding X .

We use this same notion for computing solutions to problems. Simply stated, given an algorithm for computing X , we use this algorithm for X in the algorithm for computing Y . We specifically imply that there is already a known algorithm for computing X . Not only does this say that Y is no harder to compute than X , but that in solving Y , we do not need to solve X again. Also, by using the algorithm for X , we may have made the solution to Y much easier. It may also be that if a better algorithm for X is found, there will be a better algorithm for Y .

This notion is used in the theory of parallel computation. Anderson gives an algorithm for the maximal path problem that uses an algorithm for matching [7]. Although the underlying model of computation is the PRAM, it is an example of the reduction paradigm. An interesting point about this algorithm is the fact that the known algorithm

for matching uses randomness. This makes the algorithm for maximal path use randomness. If a deterministic algorithm for matching was found with the same performance, then the algorithm for maximal path would be deterministic.

There are CTA algorithms using the reduction paradigm. The topological sort algorithm in Chapter 2 is a good example. Recall that the problem is solved by transforming the graph into vertices with level numbers. The vertices, when sorted by level numbers, will be in topological order. This changed the topological sort problem into a numerical sort problem for which there are known algorithms. Also, the computation of the level numbers is performed using matrix multiplication.

There are several points of interest in this algorithm. First, the time and resources of the matrix multiply algorithm dominated the entire algorithm. For the fastest solution, the matrix multiply algorithm required $O(n^3)$ processes while the remaining part of the algorithm required only $O(n^2)$ processes. For the solution where the matrix multiply algorithm was limited to $O(n^2)$ processes, it dominated the time, taking $O(n)$ time.

This algorithm also displayed two types of reduction. The first type, shown in the use of the matrix multiply, is where the algorithm for Y uses that algorithm for X many times. The second type, shown in the use of the sort algorithm, is where the algorithm for Y transforms the input into an instance of the problem for X . The algorithm for X needs to be used only once and after the algorithm for Y has completed its work.

Schwartz [76] gives two algorithms for the shuffle-exchange Ultracomputer using reduction. He shows how matrix inversion of a lower triangular matrix can be done using matrix multiply [68, 75]. For general matrices, he shows how to change the general matrix into a lower triangular matrix [20, 73] and use the algorithm for inversion of lower triangular matrices. The other algorithm is for connected components and uses matrix multiply for repeated squarings.

Dekel, Nassimi and Sahni [23] also use reduction in several algorithms. These include all-pairs shortest-paths, several spanning tree problems, and topological sort. The algorithms used to solve these problems include matrix multiply, sorting and several

of the graph algorithms just given.

5.2. Arbitrary Communication Algorithms

Most of the algorithms we have discussed have communication patterns that are independent of the input data. The communication structure is known before the algorithm is executed. The one exception is the Hough transform given in Chapter 4. The communication variance is displayed only in when to shift the data up or down in a column. The complete graph for all possible inputs is still a bounded degree graph, namely the torus.

There are problems for which known solutions may require any process to communicate with an other process, depending on the input. An example of this kind of problem is the maximum flow problem. Both known parallel solutions [35,79] use either a PRAM or a distributed network where all processors can communicate directly with any other processor. If these algorithms are to be used on a CTA machine, some method of dealing with the arbitrary communication must be used.

One method of dealing with these algorithms is to use the techniques for PRAM simulation on a CTA. Upfal and Wigderson [90] describe a scheme where the memory of a n processor PRAM is distributed through the local memories of a n processor CTA. It keeps multiple copies of each "shared memory cell." A read accesses a majority of the copies in order to know the correct value. A write needs to update a majority of the copies. This is shown to simulate a T step PRAM program using $O(T(\log n \log \log n)^2)$ steps of a shuffle-exchange Ultracomputer [76]. This method works when the number of shared memory locations is larger than the number of processors.

Nassimi and Sahni [63] show a different method for deterministic simulation of a n processor PRAM by a n processor CTA. This works for n shared memory locations, distributed one to a processor. They provide a Random Access Read (RAR) and a Random Access Write (RAW) based on sorting. For the RAR, requests for shared locations are sorted using the processor number of the requested data as the key. Duplicate

requests are reduced to a single request placed at the processor containing the data. The data values are then sent to the requesting processor by duplicating the necessary data and sorting the results using the requesting processor number as the key. A similar method is used for the RAW. The times are $O(\log^2 n)$ for the RAR and $O(\log^2 n + d \log n)$ for the RAW writing at most d elements into a single shared memory location. These results work for the shuffle-exchange Ultracomputer.

Upfal [89] has shown a probabilistic algorithm for simulating a n processor PRAM with a n processor shuffle-exchange Ultracomputer. The number of shared memory locations can be larger than n . The shared memory locations are randomly distributed among the n processors. It is shown that a T step PRAM program is simulated within $O(T \log^2 n)$ with "overwhelming probability."

Another method used to solve the arbitrary communication problem is to concentrate on routing messages between processors. It is obvious that an arbitrary permutation can be achieved in $O(\log^2 n)$ on a n -cube by using sorting. What we would like is a faster routing technique, as fast as $O(\log n)$. Borodin and Hopcroft [14] have looked at this question and have showed that in an interconnection structure with degree d , the time required in the worst case by any oblivious routing strategy is $\Omega(\frac{\sqrt{n}}{d^{3/2}})$. Oblivious means that the routing strategy is based only on the origin, destination, and the processor making routing decisions. Several people have given probabilistic algorithms for routing that take $O(\log n)$ expected time [71, 88, 91]

CHAPTER 6

Summary and Further Research

In this dissertation we have studied several programming paradigms for non-shared memory parallel computers. These paradigms are important, not only because they are the first step in developing a comprehensive "tool kit" of parallel programmers and they provide experience and knowledge transfer from earlier problem solutions, but they help us understand the commonalities of algorithms in each paradigm.

One of these commonalities is communication patterns. Since efficient and effective communication is the problem in parallel computer architecture, this identification of common communication patterns is important. In our study of paradigms, by focusing on algorithms and their natural communication structures, we have identified some common communication patterns. These results can provide useful data to computer architects on what communication topologies are most important.

Another commonality is the applicability of contraction techniques on multiple algorithms in the same paradigm. These contraction techniques focus on communication volume between processors, minimizing the amount of interprocessor communication. These contraction results provide the programmer of real parallel computers tools for solving real problems on today's machines.

The paradigms that we studied were compute-aggregate-broadcast, divide-and-conquer, pipelining, and reduction. Of these only the compute-aggregate-broadcast paradigm was new, but we showed how the others were applicable to non-shared memory parallel computers.

The compute-aggregate-broadcast paradigm is used in a wide variety of algorithms displaying several variations. These algorithms cover both numerical and non-numerical computation. Due to the aggregation and broadcast phases, tree

interconnection structures are used in all these algorithms. In most of these algorithms, all processes were part of the same tree, implementing global aggregate and broadcast, but local aggregate and broadcast utilizing several local trees was observed. Although the compute, aggregate, and broadcast phases are usually performed in that sequence, the variations of the paradigm displayed different phases being used first. Contraction of CAB algorithms focused on the tree. We proved that contraction of trees based on Leiserson's tree layout is optimal.

As part of the compute-aggregate-broadcast paradigm, we presented a new algorithm for topological sort. This algorithm used the ABC variation of the paradigm with non-global aggregate and broadcast. The time for the algorithm is $O(\log^2 n)$ with $O(n^3)$ processors and $O(n \log n)$ with $O(n^2)$ processors.

The divide-and-conquer algorithms in parallel computation displayed a comparable usefulness to that observed in the sequential paradigm. The parallelism is utilized in two places, namely in the divide and combine operations and the parallel solution of sub-problems. Several algorithms used the binary n-cube as the natural interconnection structure. This was the result of the divide or combine using corresponding elements in each of the sub-problems. The mesh interconnection was also shown to be used in some divide-and-conquer algorithms. And finally, while some parallel divide-and-conquer algorithms are not parallelizations of optimal sequential algorithms, there are some sequential divide-and-conquer algorithms that parallelize easily. Contraction of divide-and-conquer algorithms showed that different algorithms required different contractions of the binary n-cube to give good results.

As part of the divide-and-conquer paradigm, we presented a new algorithm for matrix multiplication. It is based on Strassen's decomposition of an $n \times n$ matrix into $4 \frac{n}{2} \times \frac{n}{2}$ matrices. This algorithm runs in time $O(n)$ using n^2 processors and time $O(\log n)$ using $\frac{n^3}{2}$ processors.

The pipelining paradigm encompassed many useful parallel algorithms. Because of the systolic algorithms, pipelining may be the most widely used technique in parallel algorithms. Not only does the pipelining paradigm include the systolic algorithms, but it includes non-systolic algorithms such as the funneled pipelines. It also includes tree algorithms using pipelining that may not be systolic due to the violation of the locality of communication requirement. For contraction of pipeline algorithms we looked at the mesh interconnection structure. We saw that coalescing was better than projection. We conjectured that this was the best contraction due to the fact that it had the smallest perimeter for a given area.

We briefly looked at the reduction technique. This technique, often used by theoreticians, can provide reasonable solutions to problems. Combined with other techniques, as was seen in the topological sorting algorithm for Chapter 2, it is very useful.

In this study, we have seen these paradigms displayed at several levels of computation. The pipelining and systolic algorithms often are implemented in special hardware. We also saw a hardware implementation of a CAB algorithm. Algorithms like the divide-and-conquer matrix multiply show the usefulness of these paradigms at the data operation level, sometimes called "fine grain parallelism". And finally, we have seen the paradigms used at the higher level of algorithm design in applications of the reduction technique and the CAB nature of parallel expert systems.

There is much to be done in the area of parallel programming paradigms. The paradigms presented here are not expected to be exhaustive. There are several known algorithms which do not seem to fit into any of these paradigms. As part of the search for more paradigms, known algorithms will be classified. Also, there may be other paradigms useful under other models of computation that do not apply to non-shared memory models.

We presented these paradigms using a description augmented with example algorithms. This is an imprecise method of defining paradigms. Although this method is usable, there is much room for research in how to express paradigms.

For parallel algorithms, besides the classification already mentioned, we expect that these paradigms will yield dividends, both in understanding and in the development of new algorithms. Already, the understanding of these paradigms have yielded new algorithms. The best example is the topological sort algorithm presented in Chapter 2.

The area of contraction contains many unknowns. First of all, we conjectured that coalescing is the best contraction for meshes. This conjecture needs to be proved. Also, we considered only three interconnection structures in contraction, the tree, the mesh and the n-cube. Although we gave tools for comparing contractions, it is possible that other interconnection structures are admissible to general contraction results.

As far as the tools that we presented, they are still very rudimentary. They are applicable in a restricted class of algorithms. Better tools are needed to cover more general conditions. As part of that, what other features of an algorithm should be considered? We concentrated on the volume of processor to processor communication because, to date, communication time is a dominant factor in execution times. Communication volume can be considered a "first order" effect. How much can be gained by considering "second order" effects? In Chapter 1 we saw how message dependence became a "first order" effect in an algorithm with a low degree of parallelism. How much does message dependence effect algorithms with high degrees of parallelism? Other things that could qualify as "second order" effects include computation between messages and buffer sizes.

Finally, we briefly touched on multi-phase contraction. Much is yet unknown about it. We mentioned three approaches to multi-phase contraction. Is one of them the best approach? If not, when are they preferable to the others? Are there other approaches to multi-phase contraction?

References

1. L. M. Adams, *Iterative Algorithms for Large Sparse Linear Systems on Parallel Computers*, Ph.D. Dissertation, University of Virginia, Charlottesville, 1982.
2. L. M. Adams and H. F. Jordan, "Is SOR colorblind?", *SIAM Journal on Scientific and Statistical Computing* 7(2):490-506, Apr. 1986.
3. A. V. Aho, J. E. Hopcroft and J. D. Ullman, *The Design and Analysis of Computer Algorithms*, Addison-Wesley, Reading, MA, 1974.
4. S. Ahuja, N. Carriero and D. Gelernter, "Linda and Friends", *Computer* 19(8):26-34, IEEE Computer Society Press, Aug. 1986.
5. M. Ajtai, J. Komlos and E. Szemerédi, "An $O(n \log n)$ Sorting Network", *Proceeding of the Fifteenth Annual ACM Symposium on Theory of Computing*, 1-9, 1983.
6. J. R. Allen, K. Kennedy, C. Porterfield and J. Warren, "Conversion of Control Dependencies to Data Dependencies", *Proceedings of the 10th Symposium on Principles of Programming Languages*, 177-189, 1983.
7. R. Anderson, *The Complexity of Parallel Algorithms*, Ph.D. Dissertation, Stanford University, 1985.
8. J. Baer, *Computer Systems Architecture*, Computer Science Press, Potomac, Maryland, 1980.
9. D. A. Bailey and J. E. Cuny, "An Efficient Embedding of Large Trees in Processor Grids", *Proceedings of the 1986 International Conference on Parallel Processing*, 819-822, 1986.
10. K. E. Batcher, "Sorting Networks and their Applications", *Proceedings of the AFIPS Spring Joint Computer Conference* 32 :307-314, 1968.
11. F. Berman and L. Snyder, "On Mapping Parallel Algorithms into Parallel Architectures", *Proceedings of the 1984 International Conference on Parallel*

Processing, 307-309, 1984.

12. F. Berman, M. Goodrich, C. Koelbel, W. J. Robison III and K. Showell, "Prep-P: A Mapping Preprocessor for CHiP Architectures", *Proceedings of the 1985 International Conference on Parallel Processing*, 731-733, 1985.
13. S. H. Bokhari, "On the Mapping Problem", *IEEE Transactions on Computers* C-30(3):207-214, Mar. 1981.
14. A. Borodin and J. E. Hopcroft, "Routing, Merging and Sorting on Parallel Models of Computation", *Proceedings of the Fourteenth Annual ACM Symposium on Theory of Computing*, 338-344, 1982.
15. R. H. Campbell and R. B. Kolstad, "An Overview of Path Pascal's Design", *SIGPLAN Notices* 15(9):13-14, Sep. 1980.
16. T. C. Chen, "Overlap and Pipeline Processing", in *Introduction to Computer Architecture*, H. S. Stone (editor), Science Research Associates, Chicago, 427-485, 1980.
17. S. A. Cook, "Towards a Complexity Theory of Synchronous Parallel Computation", *L'Enseignement Mathematique* XXVII :99-124, 1981.
18. S. A. Cook, "A Taxonomy of Problems with Fast Parallel Algorithms", *Information and Control* 64 :2-22, Academic Press, 1985.
19. W. Crowther, J. Goodhue, E. Starr, R. Thomas, W. Milliken and T. Blackadar, "Performance Measurements on 128-node Butterfly Parallel Processor", *Proceedings of the 1985 International Conference on Parallel Processing*, 531-540, 1985.
20. L. Csanky, "Fast Parallel Matrix Inversion Algorithms", *SIAM Journal on Computing* 5(4):618-623, Nov. 1976.
21. J. E. Cuny, Private Communication, Oct. 1986.
22. R. E. Cypher and J. L. C. Sanz, "The Hough Transform Has $O(N)$ Complexity on SIMD $N \times N$ Mesh Array Architectures", Technical Report 87-03-04, University

of Washington, Seattle, Mar. 1987.

23. E. Dekel, D. Nassimi and S. Sahni, "Parallel Matrix and Graph Algorithms", *SIAM Journal on Computing* 10(4):657-675, Nov. 1981.
24. E. Dekel and S. Sahni, "Parallel Scheduling Algorithms", *Proceedings of the 1981 International Conference on Parallel Processing*, 350-351, 1981.
25. R. J. Douglass, "A Qualitative Assessment of Parallelism in Expert Systems", *IEEE Software* 2(3):70-81, May 1985.
26. R. O. Duda and P. E. Hart, "Use of the Hough Transformation To Detect Lines and Curves in Pictures", *Communications of the ACM* 15(1):11-15, Jan. 1972.
27. P. W. Dymond and S. A. Cook, "Hardware Complexity and Parallel Computation", *Proceedings of the 21st Annual Symposium on Foundations of Computer Science*, 360-372, 1980.
28. C. Ebeling, *All the Right Moves: A VLSI Architecture for Chess*, Ph.D. Dissertation, Carnegie-Mellon University, Pittsburgh, PA, 1986.
29. F. Fich, P. L. Ragde and A. Wigderson, "Relations Between Concurrent-Write Models of Parallel Computation", *Proceedings of the 3rd Annual ACM Symposium on Principles of Distributed Computing*, 179-189, 1984.
30. G. Fielland and D. Rogers, "32-bit Computer System Shares Load Equally Among up to 12 Processors", *Electronic Design* 32(18):153-168, Sep. 6, 1984.
31. C. L. Forgy, *On the Efficient Implementation of Production Systems*, Ph.D. Dissertation, Carnegie-Mellon University, 1979.
32. S. Fortune and J. Wyllie, "Parallelism in Random Access Machines", *Proceedings of the Tenth Annual ACM Symposium on Theory of Computing*, 114-118, 1978.
33. D. Gajski, D. Kuck, D. Lawrie and A. Sameh, "Cedar -- a Large Scale Multiprocessor", *Proceedings of the 1983 International Conference on Parallel Processing*, 524-529, 1983.

34. D. Gannon, L. Snyder and J. Van Rosendale, "Programming Substructure Computations for Elliptic Problems on the CHiP System", in *Impact of New Computing Systems on Computational Mechanics*, A. K. Noor (editor), The American Society of Mechanical Engineers, 65-80, 1983.
35. A. V. Goldberg and R. E. Tarjan, "A New Approach to the Maximum Flow Problem", *Proceedings of the Eighteenth Annual ACM Symposium on Theory of Computing*, 136-146, 1986.
36. L. M. Goldschlager, "A Unified Approach to Models of Synchronous Parallel Machines", *Proceedings of the Tenth Annual ACM Symposium on Theory of Computing*, 105-111, 1978.
37. A. Gottlieb, B. D. Lubachevsky and L. Rudolph, "Basic Techniques for the Efficient Coordination of Very Large Numbers of Cooperating Sequential Processors". *ACM Transactions on Programming Languages and Systems* 5(2):164-189, Apr. 1983.
38. L. J. Guibas, H. T. Kung and C. D. Thompson, "Direct VLSI Implementation of Combinatorial Algorithms", *Proceedings of the Conference on Very Large Scale Integration: Architecture, Design, Fabrication*, Cal Tech, 255-264, Jan. 1979.
39. A. Gupta, "Implementing OPS5 Production Systems on DADO", *Proceedings of the 1984 International Conference on Parallel Processing*, 83-91, 1984.
40. F. Hayes-Roth, "Knowledge-Based Expert Systems", *Computer* 17(10):263-273, Oct. 1984.
41. C. A. R. Hoare, "Quicksort", *Computer* 5(1):10-15, Jan. 1962.
42. C. A. R. Hoare, "Communicating Sequential Processes", *Communications of the ACM* 21(8):666-677, Aug. 1978.
43. P. H. Hochschild, E. W. Mayr and A. R. Siegel, "Techniques for Solving Graph Problems in Parallel Environments", *Proceedings of the 1983 International Conference on Parallel Processing*, 351-359, 1983.

44. J. E. Hopcroft and J. D. Ullman, *Introduction to Automata Theory, Languages, and Computation*, Addison Wesley, Reading, MA, 1979.
45. E. Horowitz and A. Zorat, "Divide-and-Conquer for Parallel Processing", *IEEE Transactions on Computers* C-32(6):582-585, June 1983.
46. E. Horowitz and S. Sahni, *Fundamentals of Computer Algorithms*, Computer Science Press, Rockville, MD, 1984.
47. C. Hsiao, *Highly Parallel Processing of Relational Databases*, Ph.D. Dissertation, Purdue University, Dec. 1982.
48. INMOS Limited, *OCCAM Programming Manual*, Prentice-Hall, Englewood Cliffs, NJ, 1984.
49. A. Kapaun, K. Wang, D. Gannon, J. Cuny and L. Snyder, "The Pringle: an Experimental System for Parallel Algorithm and Software Testing", *Proceedings of the 1984 International Conference on Parallel Processing*, 1-6, 1984.
50. R. M. Keller, "Comment in a technical presentation", *Parallel Architectures Workshop, Boulder, Colorado*, 1982.
51. D. J. Kuck, R. H. Kuhn, B. Leasure and M. Wolfe, "The Structure of an Advanced Vectorizer for Pipelined Processors", *Proceedings of the 4th International Computer Software Applications Conference*, 709-715, 1980.
52. H. T. Kung and C. E. Leiserson, "Systolic Arrays (for VLSI)", in *Sparse Matrix Proceedings 1978*, I. S. Duff and G. W. Stewart (editors), Society for Industrial and Applied Mathematics, 256-282, 1979.
53. H. T. Kung, "Why Systolic Architectures?", *Computer* 15(1):37-46, Jan. 1982.
54. S. Y. Kung, K. S. Arun, R. J. Gal-Ezer and D. B. B. Rao, "Wavefront Array Processor: Language, Architecture, and Applications", *IEEE Transactions on Computers* C-31(11):1054-1065, Nov. 1982.
55. H. T. Kung, *A Listing Of Systolic Papers*, Department of Computer Science, Carnegie-Mellon University, Pittsburgh, Pennsylvania, Sep. 1986.

56. R. Ladner, *Complexity Theory, Class Notes*, University of Washington, Seattle, WA, Jan. 1982.
57. F. T. Leighton, *Layouts for the Shuffle-Exchange Graph and Lower Bound Techniques for VLSI*, Ph.D. Dissertation, MIT, Aug. 1981.
58. C. E. Leiserson, *Area-Efficient VLSI Computation*, MIT Press, Cambridge, MA, 1983.
59. H. F. Li and R. Jayakumar, "Systolic Structures: A Notion and Characterization", *Journal of Parallel and Distributed Computing* 3(3):373-397, Academic Press, Sep. 1986.
60. J. D. Lipson, *Elements of Algebra and Algebraic Computing*, Addison-Wesley, Reading, MA, 1981.
61. J. R. McGraw, "The VAL Language: Description and Analysis", *ACM Transactions on Programming Languages and Systems* 4(1):44-82, Jan. 1982.
62. D. Nassimi and S. Sahni, "Finding Connected Components and Connected Ones on a Mesh-Connected Parallel Computer", *SIAM Journal on Computing* 9(4):744-757, Nov. 1980.
63. D. Nassimi and S. Sahni, "Data Broadcasting in SIMD Computers", *IEEE Transactions on Computers* C-30(2):101-107, IEEE, Feb. 1981.
64. P. A. Nelson, "A Non-systolic Matrix Product Algorithm", Technical Report 85-11-02, Department of Computer Science, University of Washington, Nov. 1985.
65. P. A. Nelson and L. Snyder, "Programming Solutions to the Algorithm Contraction Problem", *Proceedings of the 1986 International Conference on Parallel Processing*, 258-261, Aug. 1986.
66. P. A. Nelson and L. Snyder, "Programming Paradigms for Nonshared Memory Parallel Computers", in *The Characteristics of Parallel Algorithms*, L. Jamieson, D. Gannon and R. Douglass (editors), MIT Press, Cambridge, MA, 3-20, 1987.

67. N. Nilsson, *Principles of Artificial Intelligence*, Tioga Press, Palo Alto, CA, 1982.
68. S. E. Orcutt, "Parallel Solution Methods for Triangular Linear Systems of Equations", Technical Report 77, Digital Systems Laboratory, Stanford University, 1974.
69. M. S. Paterson, W. L. Ruzzo and L. Snyder, "Bounds on the Minimax Edge Length for Complete Binary Trees", *Proceedings of the Thirteenth Annual ACM Symposium on Theory of Computing*, 293-229, 1981.
70. G. F. Pfister, W. C. Brantley, D. A. George, S. L. Harvey, W. J. Kleinfelder, K. P. McAuliffe, E. A. Melton, V. A. Norton and J. Weiss, "The IBM Research Parallel Processor Prototype (RP3): Introduction and Architecture", *Proceedings of the 1985 International Conference on Parallel Processing*, 764-771, 1985.
71. N. Pippenger, "Parallel Communication with Limited Buffers", *Proceedings of the 25th Symposium on Foundations of Computer Science*, 127-136, 1984.
72. V. K. Prasana Kumar and M. M. Eshaghian, "Parallel Geometric Algorithms for Digitized Pictures on Mesh of Trees", *Proceedings of the 1986 International Conference on Parallel Processing*, 270-273, 1986.
73. F. P. Preparata and D. V. Sarwate, "An Improved Parallel Processor Bound in Fast Matrix Inversion", *Information Processing Letters* 7(3):148-150, Apr. 1978.
74. F. P. Preparata and J. Vuillemin, "The Cube-Connected Cycles: A Versatile Network for Parallel Computation", *Communications of the ACM* 24(5):300-309, May 1981.
75. A. Sameh and R. Brent, "Solving Triangular Systems on a Parallel Computer", *SIAM Journal on Numerical Analysis* 14(6):1101-1113, Dec. 1977.
76. J. T. Schwartz, "Ultracomputers", *ACM Transactions on Programming Languages and Systems* 2(4):484-521, Oct. 1980.
77. R. Sedgewick, *Algorithms*, Addison Wesley, Reading, MA, 1893.

78. C. L. Seitz, "The Cosmic Cube", *Communications of the ACM* 28(1):22-33, Jan. 1982.
79. Y. Shiloach and U. Vishkin, "An $O(n^2 \log n)$ Parallel Max-Flow Algorithm", *Journal of Algorithms* 3(2):128-146, Academic Press, Inc., June 1982.
80. L. Snyder, "Introduction to the Configurable, Highly Parallel Computer", *Computer* 15(1):47-56, Jan. 1982.
81. L. Snyder, "Parallel Programming and the Poker Programming Environment", *Computer* 17(7):27-36, July 1984.
82. L. Snyder, "Type Architectures, Shared Memory and the Corollary of Modest Potential", *Annual Review of Computer Science*, 1986.
83. L. Snyder, "Poker (4.0) Programmer's Reference Guide", Technical Report 86-05-04, Computer Science Department, University of Washington, 1987.
84. S. J. Stolfo and D. P. Miranker, "DADO: A Parallel Processor for Expert Systems", *Proceedings of the 1984 International Conference on Parallel Processing*, 74-82, 1984.
85. Q. F. Stout, "Supporting Divide-and-Conquer Algorithms for Image Processing", *Journal of Parallel and Distributed Computing* 4(1):95-115, Academic Press, Feb. 1987.
86. V. Strassen, "Gaussian elimination is not optimal", *Numerische Mathematik* 13 :354-356, 1969.
87. W. Su, R. Faucette and C. L. Seitz, "C Programmer's Guide to the Cosmic Cube", Technical Report 5203:TR:85, Computer Science Department, California Institute of Technology, Sep. 1985.
88. E. Upfal, "Efficient Schemes for Parallel Communication", *Journal of the ACM* 31(3):507-517, July 1984.
89. E. Upfal, "A Probabilistic Relation Between Desirable and Feasible Models of Parallel Computation", *Proceedings of the Sixteenth Annual ACM Symposium on*

Theory of Computing, 258-265, 1984.

90. E. Upfal and A. Wigderson, "How to Share Memory in a Distributed System", *Proceedings of the 25th Symposium on Foundations of Computer Science*, 171-180, 1984.
91. L. G. Valiant and G. J. Brebner, "Universal Schemes for Parallel Communication", *Proceedings of the Thirteenth Annual ACM Symposium on Theory of Computing*, 263-277, 1981.
92. J. L. Wagener, "Status of Work Toward Revision of Programming Language Fortran", *SIGNUM Newsletter* 19(3):5-42, July 1984.
93. L. C. Widdoes, Jr., "The S-1 Project: Developing High-Performance Digital Computers", *Proceedings of the IEEE Compcom*, 282-291, 1980.
94. D. Young, *Iterative Methods for Solving Partial Differential Equations of Elliptic Type*, Ph.D Dissertation, Harvard University, Cambridge, MA, 1950.
95. D. Young, *Iterative Solutions of Large, Linear Systems*, Academic, New York, 1971.

APPENDIX A

Implementations of Selected Algorithms

As part of this dissertation, we include the implementation of several algorithms. Although this does not contribute directly to the understanding of paradigms, seeing an implementation for an algorithm may help one understand the algorithm and give insight into the special features of a paradigm displayed in the algorithm. The algorithms implemented are all discussed in Chapters 2 through 4. As such, we do not discuss the algorithm, but we mention special points about the implementation that were not discussed in the chapters. All implementations are done using the Poker programming environment [81]. We give a description of the Poker programming environment and present the implementations of the algorithms.

A.1. The Poker Programming Environment

Poker is a programming environment designed for writing parallel programs for the CHiP architecture [80]. It represents a parallel program as a data base containing multiple phases of computation. Each phase is represented by a communications graph, an assignment of processes to processors, an assignment of logical names to actual communication channels, and process codes. These components are manipulated using a graphical interface that allows the Poker program to be displayed and edited from several views. For our purpose, it is sufficient to understand the information presented in these views and not how to interact with the environment.

As an example program, consider the implementation of the maximum algorithm given in Chapter 1. The algorithm used a 15 node tree interconnection structure with special processes at the leaves, internal nodes, and the root. Each process has a value and the result is to terminate with the maximum in the root process.

Poker works only with square arrays of processors. Each processor is numbered by both row and column. (In this example we use processor 1,3 as an example because it is not part of the tree algorithm.) The "Switch Setting View" allows the programmer to view and modify the communications graph. Figure A-1 shows a Poker representation of a 15 node tree. Notice that each processor is represented by a square, and the communication channels are represented by connections to the square. Only 8 connections to each processor is allowed. They attach to the processor at the eight compass points of north, northeast, east, southeast, south, southwest, west, and northwest. The north connection is at the "top" of the processor.

The "Code Names View" allows the programmer to specify which sequential process will be executed at each of the processors in the square. Figure A-2 shows the "Code Names View". The process *leaf* will be executed at processor 1,1. Notice the

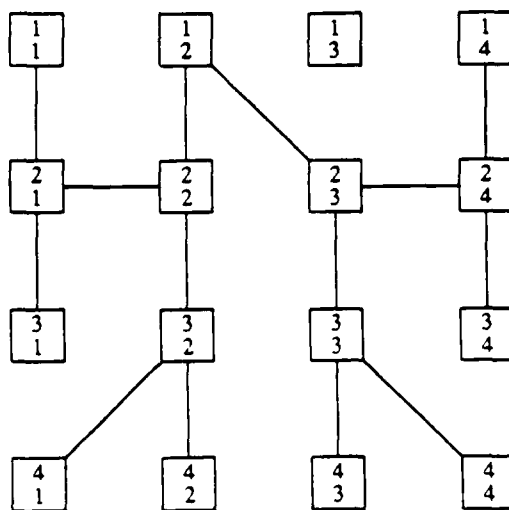


Figure A-1: A 15 Node Tree in Poker

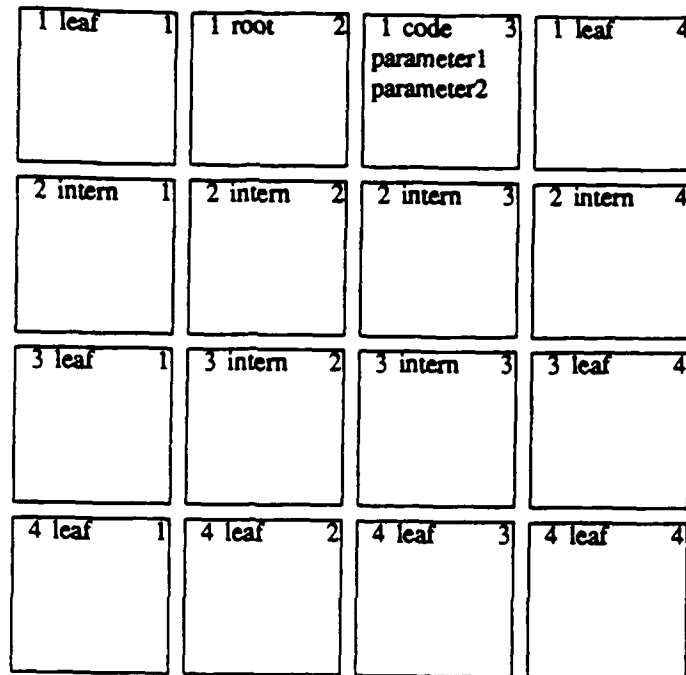


Figure A-2: Code Names View for Maximum

correspondence of the processors to the "Switch Setting View." Although it is not used for the maximum algorithm, processor 1,3 shows that parameters may be given to the processes at the start of the phase.

These processes are programmed in one of two languages, XX (dos equis) or Poker C [83]. XX is the first language supported by Poker and is used for programming the Pringle [49]. Poker C was later added for speed of simulation on a sequential processor. Both languages are used in the following implementations. The major constructs of interest for our programs appear in both languages because they are designed for use with Poker. This construct has to do with communication. Processes communicate with each other through *ports*. In the program, these ports are given logical names. Figure A-3 shows the code for a leaf processor. Notice that the only process a leaf process communicates with is a parent process. Figure A-4 shows the code for an internal processor. Notice that the internal process has ports for its left child, right child and

```

code leaf;

ports Parent;

begin

  int Value;

  Parent <- Value;

end.

```

Figure A-3: A Leaf Process

```

code internal;

ports Left, Right, Parent;

begin

  int Value;
  int max, temp;

  max := Value;
  temp <- Left;
  if temp > max then max := temp;
  temp <- Right;
  if temp > max then max := temp;

  Parent <- max;

end.

```

Figure A-4: An Internal Process

parent. For completeness Figure A-5 shows the code for the root process.

Communication with the other processors is performed by the port I/O statement. The port I/O statement has two forms:

```

variable <- port
port <- expression

```

where the first form represents reading from the port (receiving a message from the other

process) and the second form represents writing to a port.

These logical port names allow one process to be used at several processors using different communication channels for the same logical communication. The "Port Names View" allows the programmer to specify the relationship between logical port name and physical communication channel for each of the processors. Figure A-6 shows the port names for the maximum algorithm. Processor 1,3 shows the logical name placement in the square corresponding to the physical connections. Notice that processor 1,1 communicates on the south connection while processor 3,1 communicates on the north channel for the same logical name.

The last feature we must mention is how data is transferred from phase to phase. In XX, the variables in the process codes are allocated in the same place in memory, providing an unnamed common block. Values written in one phase are available in the next phase. To insure correct values, the order and types of the variables in the two process codes must be identical.

```

code root;

ports Left, Right;

begin

  int Value;
  int max, temp;

  max := Value;
  temp <- Left;
  if temp > max then max := temp;
  temp <- Right;
  if temp > max then max := temp;

end.

```

Figure A-5: The Root Process

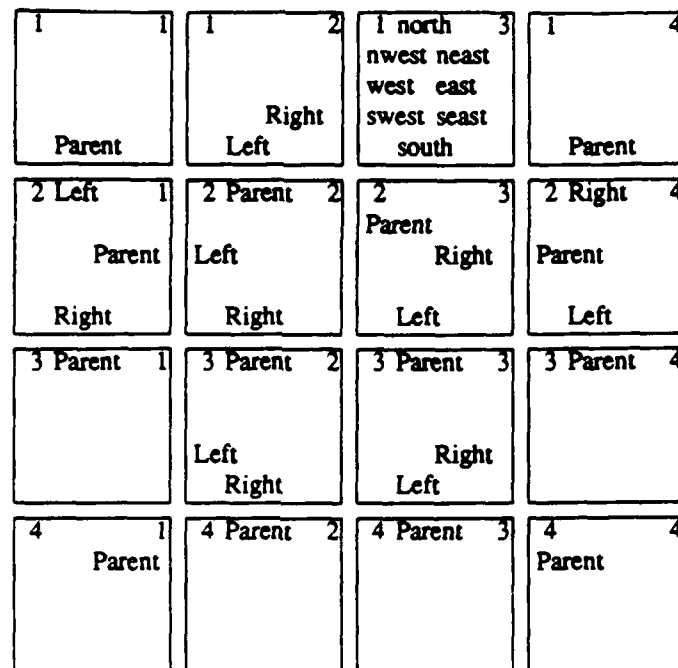


Figure A-6: Port Names View for Maximum

In Poker C, this phase to phase communication is facilitated by use of the "inter-phase variable space." The inter-phase variable space is a collection of variables accessible by all phases. To read a value from the inter-phase variable space, a process "imports" a value into a local process variable using the *import* statement. To set a value, a process "exports" an expression. The form of these statements are:

```
import(local_variable,inter-phase)
```

```
export(expression,inter-phase).
```

In the following implementations, we present enough of the Poker structures to completely define the program. In some programs, only one process code is used. For those programs, we do not include the Code Names View. For others, all processors have the same port name assignment and only a single processor is shown for the Port Names View. Codes are in both XX and Poker C, but a single program has codes in only one of the languages.

A.2. Jacobi Iterations

The following program solves the electric field problem as given in Chapter 2. The square is represented by 256 discrete points. Due to the fact that the upper right is a mirror image of the other three quadrants, using reflection and negation, only the upper right is computed. The initial guesses are the parameters to the compute phase. A special code is placed in processors representing the charged bar. The aggregation phase is a tree that does minimum sum. The control script implements the broadcast by restarting the compute-aggregate loop when the algorithm is not completed. Figures A-7 and A-8 show the switch settings for phase 1 and phase 2 respectively. Figures A-9, A-10, A-11 and A-12 show the code names and port names for enough processors to provide all the necessary information.

The following are the codes for each process and the Poker script to run the entire algorithm.

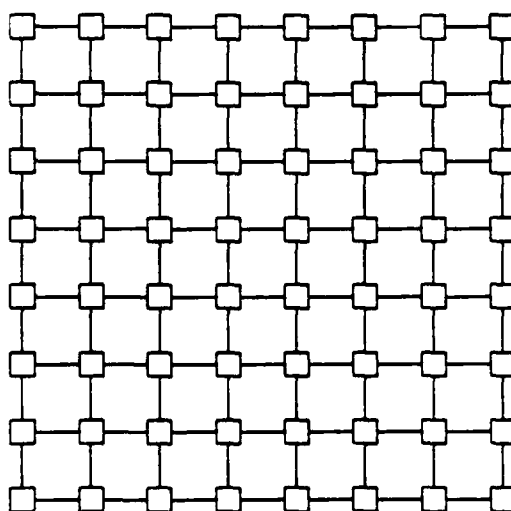


Figure A-7: Jacobi Phase 1 Switch Settings

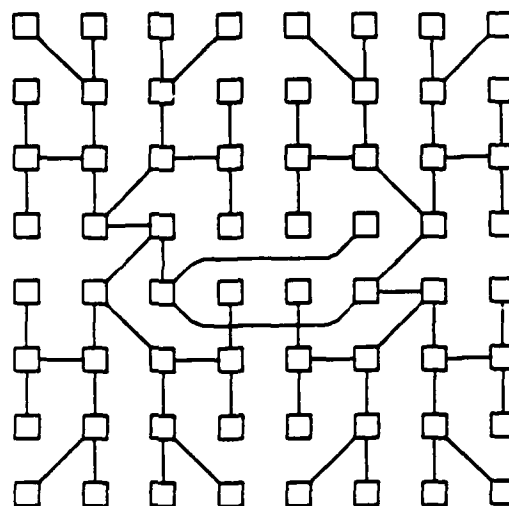


Figure A-8: Jacobi Phase 2 Switch Settings

5 var 50	5 var 50	5 var 50	5 var 50
6 const 100	6 const 100	6 const 100	6 var 50
7 var 50	7 var 50	7 var 50	7 var 50
8 var 50	8 var 50	8 var 50	8 var 50

Figure A-9: Jacobi Phase 1 Code Names

5 leaf 1	5 inode 2	5 root 3
6 inode 1	6 inode 2	6 inode 3
7 leaf 1	7 inode 2	7 inode 3

Figure A-10: Jacobi Phase 2 Code Names

1 north 1
west east
south

Figure A-11: Jacobi Phase 1 Port Names

5 parent	5 parent right left	5 left third right
6 right parent left	6 parent left right	6 parent right left
7 parent	7 parent left right	7 parent right left

Figure A-12: Jacobi Phase 2 Port Names

const.pc

code const;

ports north, east, south, west;

trace newval;

```

main (ival)
float ival;
{
    float curval, newval;
    float nval, eval, sval, wval;
    float diff;

    /* initial value/current value */
    curval = -1;
    import(curval,val);
    if (curval < 0) curval = ival;

    /* send */
    if (PEi != 1) north <- curval;
    if (PEj != PEn) east <- curval;
    if (PEi != PEn) south <- curval;
    if (PEj != 1) west <- curval;

    /* receive */
    if (PEi != 1) nval <- north;
    if (PEj != PEn) eval <- east;
    if (PEi != PEn) sval <- south;
    if (PEj != 1) wval <- west;

    /* calculate */
    newval = curval;
    diff = 0;

    /* report */
    export(newval,val);
    export(diff,diff);
}

```

var.pc

code var;

ports north, east, south, west;

trace newval;

```

main (ival)
float ival;
{

```

```

float curval, newval;
float nval, eval, sval, wval;
float diff;

/* initial value/current value */
curval = -1;
import(curval, val);
if (curval < 0) curval = ival;

/* send */
if (PEi != 1) north <- curval;
if (PEj != PEn) east <- curval;
if (PEi != PEn) south <- curval;
if (PEj != 1) west <- curval;

/* receive */
if (PEi != 1) nval <- north;
else nval = 0;
if (PEj != PEn) eval <- east;
else eval = 0;
if (PEi != PEn) sval <- south;
else sval = -curval;
if (PEj != 1) wval <- west;
else wval = curval;

/* calculate */
newval = (nval+eval+sval+wval) / 4;
diff = newval-curval;
if (diff<0) diff = -diff;

/* report */
export(newval, val);
export(diff, diff);

```

```

}

```

leaf.pc

```

code leaf;

```

```

ports parent;

```

```

main()

```

```

{

```

```

    float diff;

```

```
import(diff,diff);
parent <- diff;

}

inode.pc

code inode;

ports parent,left,right;

main()
{
    float diff;
    float lval, rval;

    import(diff,diff);
    lval <- left;
    rval <- right;
    diff = (diff>lval?diff:lval);
    diff = (diff>rval?diff:rval);
    parent <- diff;

}

root.pc

code root;
trace cont, diff;

ports third,left,right;

main(tol)
float tol;
{
    float diff;
    float lval, rval,tval;
    int cont;

    import(diff,diff);
    rval <- third;
    diff = (diff>tval?diff:tval);
    lval <- left;
    rval <- right;
    diff = (diff>lval?diff:lval);
```

```

diff = (diff>rval?diff:rval);
cont = (diff<tol?0:1);
export(diff,max);

```

```

}

```

Poker script

```

run 1 trace^x
log^x
run 2 trace^x
log^x
if 5 3 max > 0.1 skip -4^x

```

A.3. Batcher's Sort

The following program is the 64 processor divide-and-conquer sort of Batcher. We present the sort phase. This was tested with two other phases that loaded a test problem and dumped the answer. The interconnection structure is the 64 processor n-cube, shown in Figure A-13 for the upper left 16 processors. All other quadrants are reflections of the upper left quadrant. Since there is a single code, batchers.pc, we do not include a code names view. The port names for the upper left 16 processors is given in Figure A-14.

batchers.pc

```

code batchers;

ports del1, del2, del4, del8, del16, del32;

main()
{
    int element;
    port del[6];
#define UP 1
#define DOWN -1

    del[0] = del1;

```

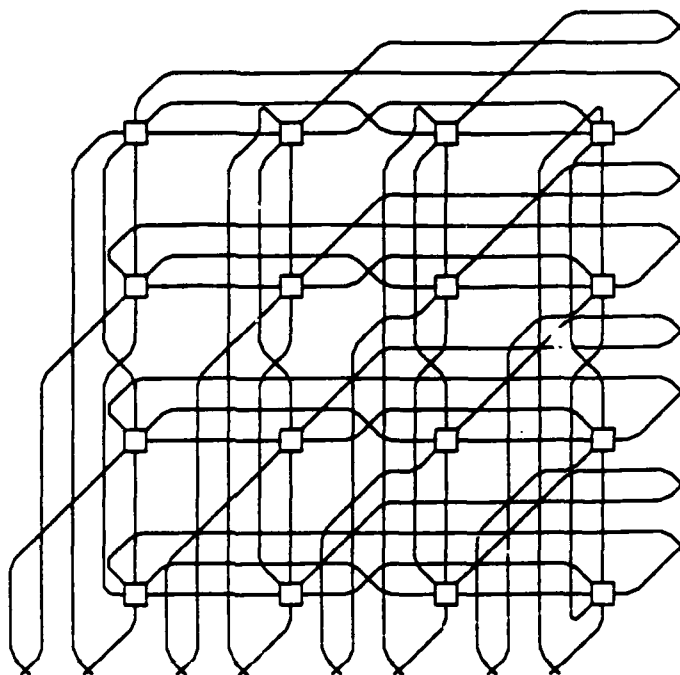


Figure A-13: 6-Cube Interconnection

```

del[1] = del2;
del[2] = del4;
del[3] = del8;
del[4] = del16;
del[5] = del32;

```

```

import(element,element);

```

```

sort(&element,PEid-1,PEn*PEn,5,UP,del);

```

```

export(element,result);

```

```

}

```

```

sort ( elem, id, n, lev, dir, del )

```

```

int *elem, id, n, lev, dir,

```

```

port del[6];

```

```

{

```

```

    if (lev >= 0) {

```

```

        n /= 2;

```

1 del4 del2 del32 del1 del16 del8	1 del32 del4 del1 del2 del16 del8	1 del32 del4 del2 del1 del16 del8	1 del32 del2 del1 del4 del16 del8
2 del8 del4 del2 del1 del32 del16	2 del8 del4 del1 del2 del32 del16	2 del8 del4 del2 del1 del32 del16	2 del8 del2 del1 del4 del32 del16
3 del16 del4 del2 del1 del32 del8	3 del16 del4 del1 del2 del32 del8	3 del16 del4 del2 del1 del32 del8	3 del16 del2 del1 del4 del32 del8
4 del8 del4 del2 del16 del1 del32	4 del8 del16 del4 del1 del2 del32	4 del8 del16 del4 del2 del1 del32	4 del8 del2 del1 del4 del16 del32

Figure A-14: Batcher's Sort Port Names

```

sort(elem,id % n, n, lev-1, id<n?UP:DOWN, del);
merge(elem,id,n,lev,dir,del);
}
}

merge ( elem, id, n, lev, dir, del )
int *elem, id, n, lev, dir,
port del[6];
{
    int temp;

    del[lev] <- *elem;
    temp <- del[lev];

    if ( (id >= n && dir == UP) || (id < n && dir == DOWN) )
        { *elem = (*elem<temp ? temp : *elem); }
    else
        { *elem = (*elem>temp ? temp : *elem); }

    if (lev > 0) merge(elem, id%n, n/2, lev-1, dir, del);
}

```

A.4. Matrix Multiply

The following program is the 64 processor divide-and-conquer matrix multiply. We present the matrix multiply phase. This was tested with two other phases that loaded a test problem and dumped the answer. The interconnection structure is the 64 processor n-cube, shown with Batcher's sort in Figure A-13. Again, since there is a single code, matmul.x, we do not include a code names view.

matmul.x

```
code matmul; /* matrix product */
```

```
trace aele, bele, cele;
```

```
ports vert2, vert4, vert8, horiz2, horiz4, horiz8;
```

```
begin
```

1 horiz8 1 horiz4 vert8 horiz2 vert4 vert2	1 vert8 horiz8 2 horiz2horiz4 vert4 vert2	1 vert8 horiz8 3 horiz4horiz2 vert4 vert2	1 vert8 horiz8 4 horiz4 horiz2horiz8 vert4 vert2
2 vert2 1 horiz8horiz4 horiz2 vert8 vert4	2 vert2 2 horiz8 horiz2horiz4 vert8 vert4	2 vert2 3 horiz8 horiz4horiz2 vert8 vert4	2 vert2 4 horiz4 horiz2horiz8 vert8 vert4
3 vert4 1 horiz8horiz4 horiz2 vert8 vert2	3 vert4 2 horiz8 horiz2horiz4 vert8 vert2	3 vert4 3 horiz8 horiz4horiz2 vert8 vert2	3 vert4 4 horiz4 horiz2horiz8 vert8 vert2
4 vert2 1 horiz8horiz4 vert4 horiz2 vert8	4 vert2 2 vert4 horiz8 horiz2horiz4 vert8	4 vert2 3 vert4 horiz8 horiz4horiz2 vert8	4 vert2 4 horiz4 horiz2horiz8 vert4 vert8

Figure A-15: DAC Matrix Multiply Port Names

```

int aele, bele, cele, othera, otherb;
int save4, save8;
sint i2, i4, i8, j2, j4, j8;

```

```

/* initialize */
cele := 0;
i2 := PEi-1;
i8 := i2 / 4 mod 2;
i4 := i2 / 2 mod 2;
i2 := i2 mod 2;
j2 := PEj-1;
j8 := j2 / 4 mod 2;
j4 := j2 / 2 mod 2;
j2 := j2 mod 2;

```

```

/* do the matrix multiply */
gosub Compute8x8;

```

```

exit; /* end of program */

```

```

/* Subroutines */

```

```

Compute2x2:
/* send values */
vert2 <- bele;
horiz2 <- aele;
otherb <- vert2;
othera <- horiz2;
/* multiply */
if i2 = j2 then
    cele := cele + aele * bele + othera * otherb
else
    cele := cele + aele * otherb + othera * bele;
return;

```

```

Compute4x4:
vert4 <- bele;
horiz4 <- aele;
if i4 = j4 then begin
    gosub Compute2x2;
    bele <- vert4;
    aele <- horiz4;
    gosub Compute2x2;
end else begin
    save4 := bele;
    bele <- vert4;

```

```

    gosub Compute2x2;
    ae1e <- horiz4;
    be1e := save4;
    gosub Compute2x2;
end;
return;

```

```

Compute8x8:
    vert8 <- be1e;
    horiz8 <- ae1e;
    if i8 = j8 then begin
        gosub Compute4x4;
        be1e <- vert8;
        ae1e <- horiz8;
        gosub Compute4x4;
    end else begin
        save8 := be1e;
        be1e <- vert8;
        gosub Compute4x4;
        ae1e <- horiz8;
        be1e := save8;
        gosub Compute4x4;
    end;
return;

```

end.

A.5. WAP Matrix Multiply

This implementation of the Wavefront Array Processor matrix multiply algorithm does not strictly follow the systolic paradigm. Instead, the data is contained in the processors to start with. The data is then circulated using the torus connections to get the same data flow as the systolic implementation. The remaining part of the algorithm is the WAP matrix multiply.

matmul.x

```

code matmul ( size );

trace ae1e, be1e, ce1e, ind1;

ports left, right, up, down;

```

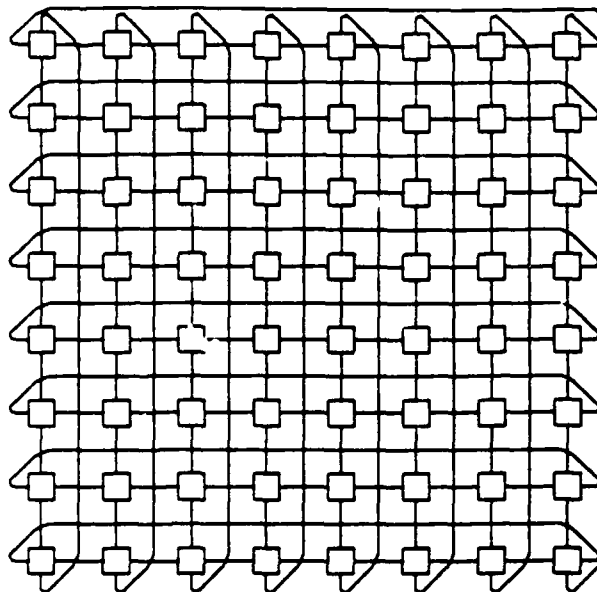


Figure A-16: WAP Matrix Multiply Interconnection

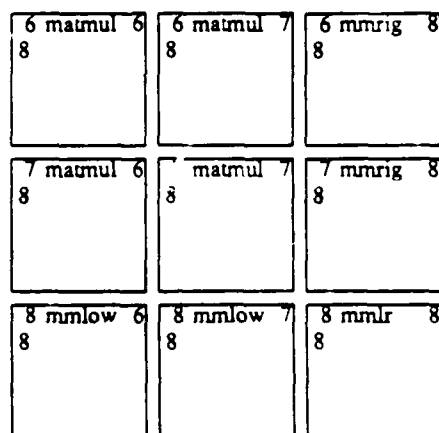


Figure A-17: WAP Matrix Multiply Code Names

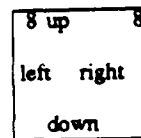


Figure A-18: WAP Matrix Multiply Port Names

```

begin

  int aele, bele, cele;
  int size;

  sint indx, PEn;

  PEn := size;
  indx := PEi;
  indx := 0;
  indx := PEj;

  /* start wave around */
  right <- aele;
  down <- bele;
  for indx := 2 to PEn do begin
    if PEj >= indx then begin aele <- left; right <- aele end;
    if PEi >= indx then begin bele <- up; down <- bele end;
  end;

  /* do the multiply */
  for indx := 1 to PEn do begin
    aele <- left;
    right <- aele;
    bele <- up;
    down <- bele;
    cele := cele + aele * bele;
  end;

end.

mmrig.x

code mmrig ( size );

```

```

trace aele, bele, cele;

ports left, right, up, down;

begin

  int aele, bele, cele;
  int size;

  sint indx, PEn;

  PEn := size;

  /* start wave around */
  right <- aele;
  down <- bele;
  for indx := 2 to PEn do begin
    if PEj >= indx then begin aele <- left; right <- aele end;
    if PEi >= indx then begin bele <- up; down <- bele end;
  end;

  /* do the multiply */
  for indx := 1 to PEn do begin
    aele <- left;
    bele <- up;
    down <- bele;
    cele := cele + aele * bele;
  end;

end.

```

mm1ow.x

```

code mm1ow ( size );

trace aele, bele, cele;

ports left, right, up, down;

begin

  int aele, bele, cele;
  int size;

```

```
sint indx, PEn;
```

```
PEn := size;
```

```
/* start wave around */
```

```
right <- aele;
```

```
down <- bele;
```

```
for indx := 2 to PEn do begin
```

```
  if PEj >= indx then begin aele <- left; right <- aele end;
```

```
  if PEi >= indx then begin bele <- up; down <- bele end;
```

```
end;
```

```
/* do the multiply */
```

```
for indx := 1 to PEn do begin
```

```
  aele <- left;
```

```
  right <- aele;
```

```
  bele <- up;
```

```
  cele := cele + aele * bele;
```

```
end;
```

```
end.
```

```
mmlr.x
```

```
code mmlr ( size );
```

```
trace aele, bele, cele;
```

```
ports left, right, up, down;
```

```
begin
```

```
  int aele, bele, cele;
```

```
  int size;
```

```
  sint indx, PEn;
```

```
  PEn := size;
```

```
/* start wave around */
```

```
right <- aele;
```

```
down <- bele;
```

```

for indx := 2 to PEn do begin
  if PEj >= indx then begin aele <- left; right <- aele end;
  if PEi >= indx then begin bele <- up; down <- bele end;
end;

```

```

/* do the multiply */
for indx := 1 to PEn do begin
  aele <- left;
  bele <- up;
  cele := cele + aele * bele;
end;

```

```

end.

```

A.6. Contractions

As part of our study of contraction, we implemented two algorithms with two different contractions. These contractions were compared with the analytical tools developed in Chapter 1 and these implementations demonstrated that these tools gave accurate predictions of their relative performance. (See the contraction sections in Chapters 2 and 4.) The implementations are included here for completeness. Although the timings given were for several different problem and machine sizes, we give only the implementation for 64 processes on 16 processors. The other sizes are straight forward changes to the given implementations.

The tree contraction algorithm is maximum and the mesh contraction algorithm is WAP matrix multiply given previously. We provide enough of the interconnection structure, port names, and code names to be able to reconstruct the program.

A.6.1. Folded Tree Algorithm

```

root.x

```

```

code root ;

```

```

trace max, lmax, rmax;

```

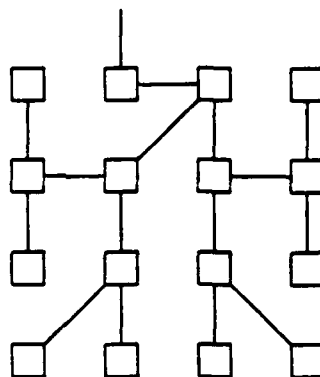


Figure A-19: Tree Folding Interconnection

1 leaf 1	1 root 2	1 intern 3	1 leaf 4
2 intern 1	2 intern 2	2 intern 3	2 intern 4
3 leaf 1	3 intern 2	3 intern 3	3 leaf 4
4 leaf 1	4 leaf 2	4 leaf 3	4 leaf 4

Figure A-20: Tree Folding Code Names

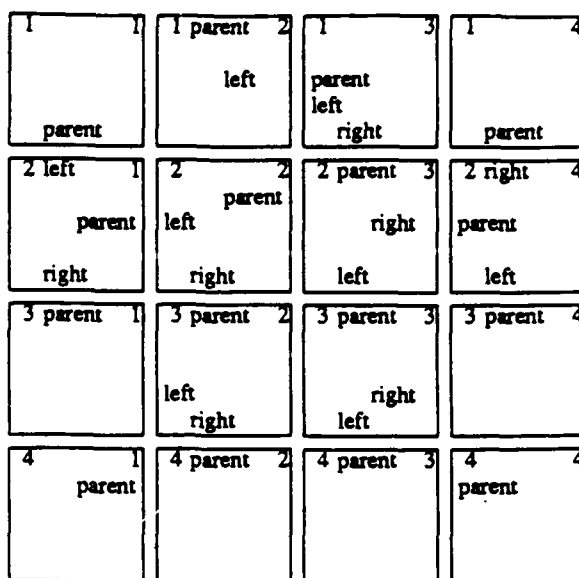


Figure A-21: Tree Folding Port Names

ports parent, left;

begin

```

int val[4];
int lval, rval, lmax, rmax, max;
sint indx;

lval <- left;
rval <- left;
if lval > rval
  then lmax := lval
  else lmax := rval;
if lmax < val[1] then lmax := val[1];
lval <- left;
rval <- left;
if lval > rval
  then rmax := lval
  else rmax := rval;
if rmax < val[2] then rmax := val[2];
if lmax > rmax
  then max := lmax
  else max := rmax;
for indx := 3 to 4 do

```

```

    if max < val[indx] then max := val[indx];
    parent <- max;

```

```

end.

```

internal.x

```

code internal ;

```

```

trace lval, rval, max;

```

```

ports parent, left, right;

```

```

begin

```

```

    int val[4];

```

```

    int lval, rval, max;

```

```

    sint indx;

```

```

    for indx := 1 to 2 do begin

```

```

        /* left tree */

```

```

        lval <- left;

```

```

        rval <- left;

```

```

        if lval > rval

```

```

            then max := lval

```

```

            else max := rval;

```

```

        if max < val[indx] then max := val[indx];

```

```

        parent <- max;

```

```

        /* right tree */

```

```

        lval <- right;

```

```

        rval <- right;

```

```

        if lval > rval

```

```

            then max := lval

```

```

            else max := rval;

```

```

        if max < val[indx+2] then max := val[indx+2];

```

```

        parent <- max;

```

```

    end;

```

```

end.

```

leaf.x

```

code leaf ;

```

```

trace indx;

```

```

ports parent;

begin

  int val[4];
  sint indx;

  for indx := 1 to 4 do parent <- val[indx];

end.

```

A.6.2. Leiserson Layout Tree Algorithm

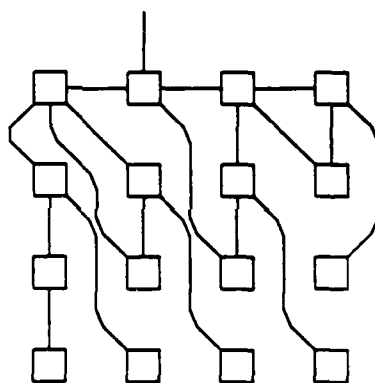


Figure A-22: Leiserson Layout Interconnection

1 inter2 1	1 inter2 2	1 inter2 3	1 inter1 4
2 inter1 1	2 inter1 2	2 inter1 3	2 leaf3 4
3 leaf3 1	3 leaf3 2	3 leaf3 3	3 leaf2 4
4 leaf1 1	4 leaf2 2	4 leaf2 3	4 leaf2 4

Figure A-23: Leiserson Layout Code Names

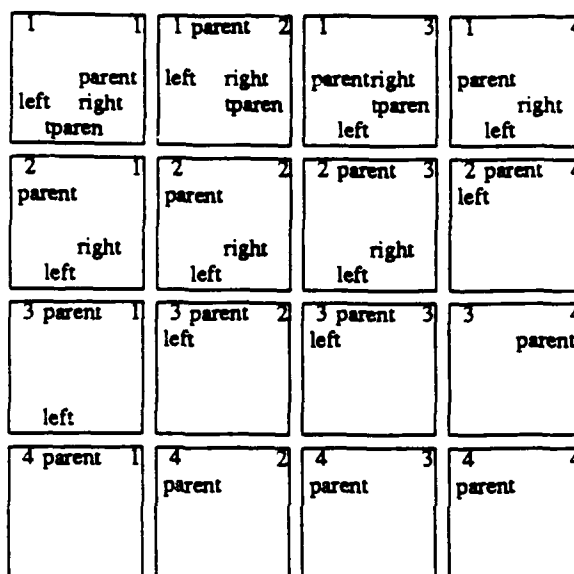


Figure A-24: Leiserson Layout Port Names

inter1.x

code inter1 ;

trace indx, max;

ports parent, left, right;

begin

int val[4];

int max, lval, rval;

sint indx;

max := val[1];

for indx := 2 to 3 do

if val[indx] > max then max := val[indx];

right <- max;

max := val[4];

lval <- left;

if lval > max then max := lval;

rval <- right;

```
    if rval > max then max := rval;  
    parent <- max;
```

```
end.
```

inter2.x

```
code inter2 ;
```

```
trace indx, max;
```

```
ports parent, left, right, tparent;
```

```
begin
```

```
    int val[4];  
    int max, lval, rval;  
    sint indx;
```

```
    max := val[1];  
    for indx := 2 to 3 do  
        if val[indx] > max then max := val[indx];
```

```
    tparent <- max;
```

```
    max := val[4];  
    lval <- left;  
    if lval > max then max := lval;  
    rval <- right;  
    if rval > max then max := rval;  
    parent <- max;
```

```
end.
```

leaf1.x

```
code leaf1 ;
```

```
trace indx, max;
```

```
ports parent;
```

```
begin
```

```
    int val[4];  
    int max;
```

```
sint indx;
```

```
max := val[1];  
for indx := 2 to 4 do  
  if val[indx] > max then max := val[indx];  
parent <- max;
```

```
end.
```

```
leaf2.x
```

```
code leaf2 ;
```

```
trace indx, max;
```

```
ports parent;
```

```
begin
```

```
int val[4];  
int max, lval;  
sint indx;
```

```
max := val[1];  
for indx := 2 to 3 do  
  if val[indx] > max then max := val[indx];
```

```
lval <- parent;  
if lval > max then max := lval;  
if val[4] > max then max := val[4];  
parent <- max;
```

```
end.
```

```
leaf3.x
```

```
code leaf3 ;
```

```
trace indx, max;
```

```
ports parent, left;
```

```
begin
```

```
int val[4];  
int max, lval;
```

```

sint indx;

max := val[1];
for indx := 2 to 3 do
  if val[indx] > max then max := val[indx];

lval <- left;
if lval > max then max := lval;
if val[4] > max then max := val[4];
parent <- max;

end.

```

A.6.3. Coalesced Mesh Algorithm

matmul.x

```

code matmul ;

trace a[1], a[2], b[1], b[2];

ports left, right, up, down;

begin

int aele[2,2], bele[2,2], cele[2,2];
int temp;

```

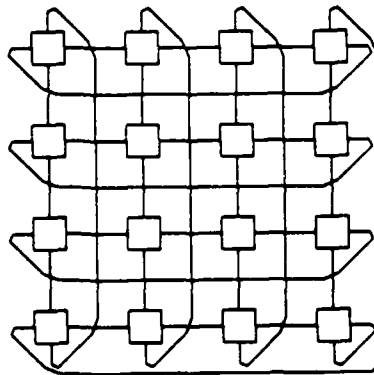


Figure A-25: Coalesced Mesh Interconnection

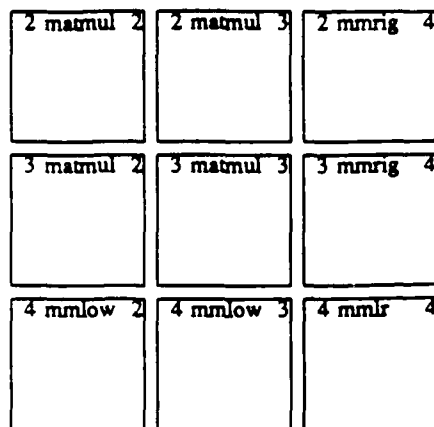


Figure A-26: Coalesced Mesh Code Names

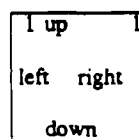


Figure A-27: Coalesced Mesh Port Names

```
int a[2], b[2];
```

```
sint indx, indx1, indx2;
```

```
/* start wave around */
```

```
indx1 := 2;
```

```
while indx1 > 0 do begin
```

```
  for indx2 := 1 to 2 do begin
```

```
    right <- aele[indx2,indx1];
```

```
    down <- bele[indx1,indx2];
```

```
  end;
```

```
  indx1 := indx1 - 1;
```

```
end;
```

```
for indx := 2 to PEn do begin
```

```
  if PEj >= indx then
```

```
    for indx1 := 1 to 4 do begin temp <- left; right <- temp end;
```

```
  if PEi >= indx then
```

```

    for indx1 := 1 to 4 do begin temp <- up; down <- temp end;
end;

```

```

for indx := 1 to 2 do
  for indx1 := 1 to 2 do
    cele[indx,indx1] := 0;

```

```

/* do the multiply */

```

```

for indx := 1 to 2*PEn do begin
  for indx1 := 1 to 2 do begin
    a[indx1] <- left; right <- a[indx1];
    b[indx1] <- up; down <- b[indx1];
  end;
  for indx1 := 1 to 2 do
    for indx2 := 1 to 2 do
      cele[indx1,indx2] := cele[indx1,indx2] + a[indx1]*b[indx2];
    end;

```

```

end.

```

mmrig.x

```

code mmrig ;

```

```

trace indx, cele[1,1], cele[1,2], cele[2,1];

```

```

ports left, right, up, down;

```

```

begin

```

```

  int aele[2,2], bele[2,2], cele[2,2];
  int temp;

```

```

  int a1, a2, b1, b2;

```

```

  sint indx, indx1, indx2;

```

```

/* start wave around */

```

```

  indx1 := 2;
  while indx1 > 0 do begin
    for indx2 := 1 to 2 do begin
      right <- aele[indx2,indx1];
      down <- bele[indx1,indx2];

```

```

    end;
    indx1 := indx1 - 1;
end;
for indx := 2 to PEn do begin
    if PEj >= indx then
        for indx1 := 1 to 4 do begin temp <- left; right <- temp end;
    if PEi >= indx then
        for indx1 := 1 to 4 do begin temp <- up; down <- temp end;
    end;

```

```

for indx := 1 to 2 do
    for indx1 := 1 to 2 do
        cele[indx,indx1] := 0;

```

```

/* do the multiply */
for indx := 1 to 2*PEn do begin
    a1 <- left;
    b1 <- up; down <- b1;
    cele[1,1] := cele[1,1] + a1*b1;
    a2 <- left;
    b2 <- up; down <- b2;
    cele[1,2] := cele[1,2] + a1*b2;
    cele[2,1] := cele[2,1] + a2*b1;
    cele[2,2] := cele[2,2] + a2*b2;
end;

```

end.

mmlow.x

code mmlow ;

trace indx, cele[1,1], cele[1,2], cele[2,1];

ports left, right, up, down;

begin

```

int aele[2,2], bele[2,2], cele[2,2];
int temp;

```

```

int a1, a2, b1, b2;

```

```

sint indx, indx1, indx2;

```

```

/* start wave around */
indx1 := 2;
while indx1 > 0 do begin
  for indx2 := 1 to 2 do begin
    right <- aele[indx2,indx1];
    down <- bele[indx1,indx2];
  end;
  indx1 := indx1 - 1;
end;
for indx := 2 to PEn do begin
  if PEj >= indx then
    for indx1 := 1 to 4 do begin temp <- left; right <- temp end;
  if PEi >= indx then
    for indx1 := 1 to 4 do begin temp <- up; down <- temp end;
end;

for indx := 1 to 2 do
  for indx1 := 1 to 2 do
    cele[indx,indx1] := 0;

/* do the multiply */
for indx := 1 to 2*PEn do begin
  a1 <- left; right <- a1;
  b1 <- up;
  cele[1,1] := cele[1,1] + a1*b1;
  a2 <- left; right <- a2;
  b2 <- up;
  cele[1,2] := cele[1,2] + a1*b2;
  cele[2,1] := cele[2,1] + a2*b1;
  cele[2,2] := cele[2,2] + a2*b2;
end;

end.

```

mmlr.x

code mmlr ;

trace indx, cele[1,1], cele[1,2], cele[2,1];

ports left, right, up, down;

begin

```
int aele[2,2], bele[2,2], cele[2,2];
int temp;
```

```
int a1, a2, b1, b2;
```

```
sint indx, indx1, indx2;
```

```
/* start wave around */
```

```
indx1 := 2;
```

```
while indx1 > 0 do begin
```

```
  for indx2 := 1 to 2 do begin
```

```
    right <- aele[indx2,indx1];
```

```
    down <- bele[indx1,indx2];
```

```
  end;
```

```
  indx1 := indx1 - 1;
```

```
end;
```

```
for indx := 2 to PEn do begin
```

```
  if PEj >= indx then
```

```
    for indx1 := 1 to 4 do begin temp <- left; right <- temp end;
```

```
  if PEi >= indx then
```

```
    for indx1 := 1 to 4 do begin temp <- up; down <- temp end;
```

```
end;
```

```
for indx := 1 to 2 do
```

```
  for indx1 := 1 to 2 do
```

```
    cele[indx,indx1] := 0;
```

```
/* do the multiply */
```

```
for indx := 1 to 2*PEn do begin
```

```
  a1 <- left;
```

```
  b1 <- up;
```

```
  cele[1,1] := cele[1,1] + a1*b1;
```

```
  a2 <- left;
```

```
  b2 <- up;
```

```
  cele[1,2] := cele[1,2] + a1*b2;
```

```
  cele[2,1] := cele[2,1] + a2*b1;
```

```
  cele[2,2] := cele[2,2] + a2*b2;
```

```
end;
```

```
end.
```

A.6.4. Folded Mesh Algorithm

The folded mesh algorithm uses the same interconnection structure and port names as the coalesced mesh. Since it uses the same code in all processors we do not give the code names for the folded mesh.

matmul.x

```
code matmul;
```

```
trace indx, indx1, indx2;
```

```
ports left, right, up, down;
```

```
begin
```

```
  int aele[2,2], bele[2,2], cele[2,2];
  int aval, bval, temp;
```

```
  sint indx, tindx;
  sint count, indx1, indx2;
```

```
  /* do this for each "data point" */
```

```
  indx1 := 1;
```

```
  indx2 := 1;
```

```
  for count := 1 to 4 do begin
```

```
    /* start wave around */
```

```
    tindx := 2;
```

```
    while tindx > 0 do begin
```

```
      right <- aele[indx1,tindx];
```

```
      down <- bele[tindx,indx2];
```

```
      for indx := 2 to PEn do begin
```

```
        if PEj >= indx then begin temp <- left; right <- temp end;
```

```
        if PEi >= indx then begin temp <- up; down <- temp end;
```

```
      end;
```

```
      tindx := tindx - 1;
```

```
    end;
```

```
  /* do the multiply */
```

```
  for indx := 1 to 2*PEn do begin
```

```
    aval <- left;
    if PEj < PEn then right <- aval;
    bval <- up;
    if PEi < PEn then down <- bval;
    cele[indx1,indx2] := cele[indx1,indx2] + aval * bval;
end;

/* next item to work on */
if count < 2 & indx1 = 1 then begin
    indx1 := indx1 + indx2;
    indx2 := 1;
end else if count > 2 & indx2 = 2 then begin
    indx2 := indx1+1;
    indx1 := 2;
end else begin
    indx2 := indx2 + 1;
    indx1 := indx1 - 1;
end;

end;

end.
```

Vita

Philip Arne Nelson [REDACTED]. After

PII Redacted

completing high school at Rio Lindo Academy in Healdsburg, California, he attended Pacific Union College in Angwin, California. In 1977 he received a Bachelor of Science with majors in Physics and Computer Science. In 1979 he received a Masters of Science in Computing Science from University of California, Davis. After teaching Computer Science for two years at Pacific Union College, he attended the University of Washington in Seattle.